DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIF 93940

11-14

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DYNAMIC LINKING IN A MICROCOMPUTER ENVIRONMENT

by

Gerald Bertram Blanton

September, 1980

Thesis Advisor: Lt.Col. R.R. Schell

Approved for public release; distribution unlimited

T195720

LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIF

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Dynamic Linking in a Microcomputer Environment		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis September, 1980
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Gerald Bertram Blanton		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE September, 1980
		13. NUMBER OF PAGES 248
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Postgraduate School Monterey, California 93940		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Dynamic linking, Linkers, Operating Systems, Microcomputers, Microprocessors, Linkers and Loaders, Dynamic Memory Management

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis presents the detailed design for a dynamic linker suitable for microcomputer operation. The design exhibits the usual property of dynamic linking in that the binding of interprocedure symbolic references to virtual addresses is deferred until the symbolic reference is first encountered during process execution. The design includes the specifications of dynamic linker modules and data structures. Furthermore, an overview of necessary operating system support is presented along with a detailed discussion of all additional translator output required. Hardware features

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
(Page 1) S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

desirable (but not necessary) in a dynamic linking environment are reviewed.

Dynamic linking without translator support and unlinking of an object (from a process address space) are investigated. A subset of the dynamic linker design (not including the unlinking capability) was implemented on an Intel 8080 microprocessor as a demonstration of the feasibility of the concepts introduced.

Approved for public release; distribution unlimited.

Dynamic Linking in a Microcomputer Environment

by

Gerald Bertram Blanton
Lieutenant, United States Navy
B.S., United States Naval Academy, 1973

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September, 1980

ABSTRACT

This thesis presents the detailed design for a dynamic linker suitable for microcomputer operation. The design exhibits the usual property of dynamic linking in that the binding of interprocedure symbolic references to virtual addresses is deferred until the symbolic reference is first encountered during process execution. The design includes the specification of dynamic linker modules and data structures. Furthermore, an overview of necessary operating system support is presented along with a detailed discussion of all additional translator output required. Hardware features desirable (but not necessary) in a dynamic linking environment are reviewed.

Dynamic linking without translator support and unlinking of an object (from a process address space) are investigated. A subset of the dynamic linker design (not including the unlinking capability) was implemented on an Intel 8080 microprocessor as a demonstration of the feasibility of the concepts introduced.

TABLE OF CONTENTS

I. INTRODUCTION -- 10

II. BACKGROUND --- 12

A. THE TRADITIONAL LINKING LOADER ------------------ 13

B. DYNAMIC LINKING ------------------------------- 14

C. OPERATING SYSTEM ENVIRONMENT FOR A DYNAMIC LINKER ---------------------------- 15

1. The Logical Levels of an Operating System --- 15

2. An Introduction to the Address Space Manager ----------------------- 17

D. TERMINOLOGY ---------------------------------- 18

III. THE LINKING PROCESS : AN OVERVIEW ------------------- 20

A. THE WALKTHROUGH ------------------------------- 21

B. A SYNOPSIS OF THE WALKTHROUGH -------------------- 25

IV. THE SPECIFICS OF DYNAMIC LINKING -------------------- 30

A. FUNCTIONS OF A LINKER -------------------------- 30

1. Snapping a Link ----------------------------- 30

a. Procedure Links ------------------------- 30

b. Data Links ----------------------------- 36

c. Construction of a List of Snapped Links - 37

2. Building Linkage Tables --------------------- 37

3. Unsnapping Objects ------------------------ 41

B. OPERATING SYSTEM SUPPORT ------------------------ 45

1. The Address Space Manager ------------------- 45

2. The Process Reference Table ------------------ 46

3. Object Deletion from a Process Address Space- 48

4. Process Initialization ---------------------- 48

C. TRANSLATOR SUPPORT -------------------------------- 49

1. External References --------------------------- 50

2. Symbolic Name Tables and Templates ---------- 50

3. Entry_Names and Entry_Points ---------------- 51

D. DYNAMIC LINKING TABLES -------------------------- 53

1. The Symbolic Name Table --------------------- 53

2. The Linkage Table ---------------------------- 55

a. The Initialized Linkage Table ----------- 55

b. Format of Snapped Links ---------------- 58

3. The Linkage Address Table ------------------ 60

E. IMPLEMENTATION OF ENTRY_NAMES AND ENTRY_POINTS -- 61

V. LINKING WITHOUT TRANSLATOR SUPPORT ------------------ 62

A. THE INTERFACE MODULE ----------------------------- 62

1. Linking of Procedures ----------------------- 63

2. Linking of Data ------------------------------ 69

B. LIMITATIONS OF UNSUPPORTED LINKING -------------- 70

VI. HARDWARE THAT ENHANCES DYNAMIC LINKING -------------- 72

A. HARDWARE FEATURES AFFECTING LINKER DESIGN ------- 72

1. Hardware Indirection and Faults on Indirection ------------------------------------ 73

2. Other Features Influencing Linker Implementation ----------------------- 74

B. HARDWARE FEATURES AFFECTING SYSTEM PERFORMANCE -- 75

VII. A DEMONSTRATION OF DYNAMIC LINKING ------------------ 77

A. THE MODULES OF THE DYNAMIC LINKER -------------- 77

        1. Process Initialization ---------------------- 80
        2. The Linker Module ---------------------------- 83
        3. The Address Space Manager Module ------------ 89
    B. THE TEST PROGRAMS ------------------------------- 92
        1. Test Program Construction ------------------- 92
            a. The Assembled Symbolic Name Table ------- 93
            b. The Assembled Template ------------------ 94
            c. Other Problems in Test Program Construction ---------------------- 95
        2. The Test Program DEMO ----------------------- 97
        3. The Test Program SUM ------------------------ 98
        4. Observations on the Implementation ---------- 104
            a. Size of the Dynamic Linker Implementation ----------------------------- 104
            b. Overhead Associated with Snapped Links -- 105
VIII. CONCLUSIONS ------------------------------------- 108
APPENDIX A - PSEUDOCODE ------------------------------- 109
            A. DYNAMIC LINKER PSEUDOCODE --------------- 109
            B. ADDRESS SPACE MANAGER PSEUDOCODE -------- 122
APPENDIX B - DEMONSTRATION SOURCE LISTING ------------- 126
            A. PROCESS INITIALIZATION SOURCE LISTING --- 126
            B. DYNAMIC LINKER MODULE SOURCE LISTING ---- 136
            C. ADDRESS SPACE MANAGER SOURCE LISTING ---- 154
            D. DISPLAY LINKAGE TABLE SOURCE LISTING ---- 172
            E. SYSTEM LIBRARY ROUTINES SOURCE LISTING -- 184

APPENDIX C - TEST PROGRAMS SOURCE LISTINGS ---------------- 197

A. DEMO SOURCE LISTING, TEMPLATE, AND RELOCATION BITS ---------------------------- 197

B. HEADER SOURCE LISTING AND TEMPLATE ------ 210

C. MULT SOURCE LISTING, TEMPLATE AND RELOCATION BITS ---------------------- 212

E. DISPLY SOURCE LISTING, TEMPLATE, AND RELOCATION BITS ---------------------- 219

F. SUM SOURCE LISTING, TEMPLATE, AND RELOCATION BITS ---------------------- 228

G. ARRAY SOURCE LISTING AND TEMPLATE ------- 237

H. OUTPUT OF DEMO --------------------------- 239

I. OUTPUT OF SUM ---------------------------- 243

LIST OF REFERENCES -- 246

INITIAL DISTRIBUTION LIST ----------------------------------- 248

LIST OF FIGURES

1. Sequence of Events for Snapping a Link to the Procedure <Target|Entry_Name> ---------- 27

2. Sequence of Events for Subsequent References to <Target|Entry_Name> ---------- 28

3. Sequence of Events for Subsequent References to <Data|Entry_Name> ---------- 29

4. Translated External Reference ---------- 34

5. Linkage Table Entries for a Snapped Procedure Link --- 35

6. Initialized Linkage Table ---------- 39

7. Process Reference Table Entry ---------- 46

8. Symbolic Name Table Entry ---------- 53

9. Data Segment Symbolic Name Table and Linkage Table --- 56

10. Data and Procedure Snapped Outgoing Links ---------- 59

11. Sequence of Events for Linking Unsupported Objects --- 64

12. Unsupported Linkage Table ---------- 68

13. The Subroutines of the Linker Module ---------- 84

14. Dynamic Linking in DEMO ---------- 99

15. Pseudocode for SUM ---------- 101

16. Object Code for SUM ---------- 102

17. Combined Linkage Table for SUM ---------- 103

I. INTRODUCTION

Dynamic linking has been previously assumed to be restricted to those computing systems that were specifically designed to support a dynamic linker. The first goal of this thesis was to determine if specialized hardware, such as found in Multics [11], is essential to realize dynamic linking. And, given that specialized hardware is not necessary, the second goal was to design a linker compatable with existing microcomputer architectures.

The design of a dynamic linker was developed with a basic set of design criteria (Table 1) established as guidelines. (A complete discussion of the implications of these criteria is delayed until the end of this thesis.) The most fundamental criterion which characterizes dynamic linking relates to when an object is bound to a virtual address within a process address space. In the traditional static environment, this binding occurs prior to program execution. In a dynamic linking environment, binding is delayed until an object is first referenced by a process. This capability allows tremendous flexibility in the development of software systems.

TABLE 1 - DESIGN CRITERIA FOR A DYNAMIC LINKER

1. Delayed Binding - The binding (linking) of an external object to a virtual address (within a process address space) must not take place until the object is first referenced during program execution.

2. Limited Overhead - Subsequent references to an object (i.e., references following the first reference) must not impose excessive overhead with respect to process execution speed and primary storage.

3. Domain Independence - The dynamic linker must be compatable with current secure operating system designs. In a multidomain environment, the dynamic linker must be capable of executing in the domain of the calling subroutine (vice executing in the security kernel[1]).

4. Syntatic Compatability - The design must allow external objects to be utilized in the same context as internally defined procedures and data. (This implies that external objects can be used as parameters subject only to the limitations of the language syntax.)

5. Pure Object Code - The dynamic linker must permit the object code of a procedure to remain pure, allowing sharing of procedures in a multiprogramming environment.

6. Hardware Independence - The design must be implementable on a microprocessor which does not possess those hardware features specifically associated with dynamic linking. In Multics [11], the features include:

   a. hardware segmentation
   b. demanding paging
   c. indirect addressing through memory
   d. a linkage fault during indirection

---

[1] It has been shown [7] that it is not necessary for the dynamic linker to reside in the security kernel to maintain system security.

II. BACKGROUND

The traditional concept of linking and loading [14] involves one, or possibly two operating system routines that load several distinct objects into memory, combine them into one address space (loading), and finally resolve addressing between objects (linking). The end result is an executable program.

The static and inflexible functions carried out by the linking loader place undesirable limitations on program development. First, a program must be intact (i.e., contain all objects required for proper execution) prior to run time. Second, if a module is changed, the whole program must be relinked. Furthermore, a module may be statically linked to several programs resulting in multiple copies of a module existing within the system. Dynamic linking is proposed as an alternative to static linking that solves these problems.

Dynamic linking [9, 11, 14] offers two other major advantages over static linking. First, dynamic linking allows a programmer to write and test incomplete programs since one may include in a subroutine a reference to an as yet unwritten external object and, as long as the reference is never executed, the program will not experience a run time error. In the field of software development, this feature is advantageous since incomplete modules may still

be tested individually. (It should be noted at this time that once the user has a completely tested product, it may be desired to statically link modules together to avoid the run time overhead associated with dynamic linking.)

The second major advantage of dynamic linking is that modules of a program need not be generated by the same translator. For example, in a dynamic linking environment one may use FORTRAN to do some double precision scientific calculations. If the results were then stored in an external data structure, they could be displayed using a dynamically linked module written in a more suitable language for I/O formatting such as PL/1. Because the modules 'communicate' via the external data structure, and are dynamically linked to each other, they need not be from the same translator. (Note that a dynamic linker does not prohibit such a 'heterogenous' program from executing but may not be sufficient in itself to allow proper execution.)

A. THE TRADITIONAL LINKING LOADER

First of all, the 'linker' and the 'loader' should be considered separate operating system functions. Linking may still be viewed as the combining of several objects into one program; however, the loading process actually consist of two distinct operations. The popular concept of a loader is one of a static operation prior to run time which takes some object code associated with a program and 'loads' this code

into main memory where it can be executed. This is the second function of a loader. The loader must first determine where each object will be placed in the address space of the process (viz., a program in execution). (This traditional concept views the address space as a linear array of memory locations.) After loading, the linking loader would link distinct objects into a single program by resolving the addressing of data and procedures defined external to individual subroutines. (It is noted that some reverse the order by linkinging loaders may link before loading).

B. DYNAMIC LINKING

The alternative to the static linking phase of the linking loader is to dynamically link separate objects at run time. This involves objects referred to in the source code of a program by a symbolic name only. The complete operation (including a dynamic linking phase) dictates that the object be located, and added to the address space of a program (i.e., assigned a virtual address). Then the reference to an object′s symbolic name is converted into an addressing instruction using the object′s virtual address. This implies that a subroutine as it exist at the beginning of run time cannot properly execute since the object code produced from a reference to a symbolic name must be converted into a virtual address in the address space of a process. This address conversion is known as dynamic

linking.

In order to support dynamic linking a system must have the ability to enter objects in the address space of a process during run time. Additionally, the operating system must be able to 'load' an object into memory during program execution. As has been noted these two functions traditionally have been considered operations associated with the loader. However, it should be apparent that this 'loading' is actually a function of dynamic memory allocation using techniques such as paging, segmentation, or dynamic relocation. Thus in a dynamic linking environment the loader functions are carried out by the operating system memory management that enters objects in a process address space.

C. OPERATING SYSTEM ENVIRONMENT FOR A DYNAMIC LINKER

1. The Logical Levels of an Operating System

It is useful at this time to propose an abstract operating system as an environment in which a dynamic linker will exist. This operating system consist of four hierarchical levels. (An operating system design along these lines has been shown feasible for microcomputers [16].) The most fundamental level consist of the hardware associated with the target machine. Above this levels is a software kernel that includes the most basic software primitives including memory management, file primitives, and

multiprocessing support. Conceptually, the kernel includes those software routines which, in a secure operating system, must be protected from malicious or inadvertant tampering. In a multiprogramming environment, the kernel provides the capability to multiplex resources (i.e., line printers, disk units, etc.) for various user processes.

The level above the kernel, the supervisor level, consist of those operating system routines which need not exist in the kernel. In general, the supervisor provides common services to all users. The final level is the user level where user programs and data reside. (It has been shown (by Jansen [7]) that the linker should be able to reside in all user levels. Jansen [7] also demonstrated that the dynamic linker need not and, more importantly, should not exist in the kernel.)

2. An Introduction to the Address Space Manager

Before an object can be linked, it must be addressable by a process. In a static environment, this would equate to loading the object in the address space of a process by allocating to it a linear block of memory. Essentially this is what is done in a dynamic environment except the object retains its identity as a distinct segment and is allocated a virtual address[2]. (In this thesis, virtual addresses will be considered to consist of a segment number plus some offset from the base of that segment.) The assignment of a virtual address to an object will be done by the address space manager.

The address space manager is invoked by the dynamic linker with a request to make an object known. The address space manager does this by assigning to the object a unique identifier, such as a segment number, that can be used to access the object within the process address space. An entry for the object will then be made by the address space manager in a table to prevent assigning multiple identifiers to the same object. This implies that a search would first be made of this table, which is called the Process Reference

---

[2] A virtual address is a potentially relocatable address which may be converted into an absolute address by hardware. It may consist of a segment number and offset, or some other relative format in which the base address of the segment is added to an offset to achieve the absolute address. (However this does not imply that segmentation hardware is necessary in a dynamic linking environment.)

Table [3], to determine if the object is already known. If not, the address space manager would have the object assigned a segment number (identifier), create an entry for the object in the process reference table, and return this segment number to the linker.

D. TERMINOLOGY

In order to ensure that the terminology used is understood, the following definitions are offered.

A subroutine will be defined as a basic unit of standalone, executable code (i.e., a procedure). Several subroutines and data objects can be combined to form a program. Stated another way, a program consist of all subroutines and data modules utilized by that program during its execution. A process [1] is a program in execution and is characterized by an execution point (usually defined by a hardware program counter) and an address space. During execution, a subroutine may call an external object that is known to that subroutine only by its symbolic name prior to execution. The reference to an external object within a subroutine will be called an external reference [15]. An external object [4] may consist of either data (external data) or an external procedure (that is itself a subroutine). Each object is a distinct logical entity and

---

[3] In Multics [11], the process reference table is called the known segment table.

will at times also be referred to as a segment [14]. (An effort is made to use the term "object" whenever possible to avoid the implication that a processor featuring hardware segmentation is necessary in a dynamic linking environment.)

III. THE LINKING PROCESS: AN OVERVIEW

Before detailing the dynamic linking process, a brief walkthrough of the steps involved in establishing a link between the subroutine <Caller> and some external procedure <Target¦Entry_Name> will be investigated. (Entry_Name represents one of multiple accesses, or entry points, into <Target>. An entry point into an object can be considered a label that can be referenced by an external object. Associated with each entry point is a unique entry name, and an entry point offset that represents the relative offset of the entry point from the starting location of the object.[4])

Fundamentally, the following events must occur to link <Target¦Entry_Name> to <Caller>. The linker must be invoked when a reference to <Target¦Entry_Name> is first encountered. The linker must be capable of accessing the symbolic name "Target¦Entry_Name" and using that symbolic name to learn the segment number of <Target>. The linker will then establish a link to <Target¦Entry_Name> such that subsequent references found in <Caller> will not require invocation of the linker but instead will result in either a call to <Target¦Entry_Name>, in the case of an external

---

[4] The term 'entry point' has evolved as representing either the label 'entry point' or the offset associated with that label [11, 14]. This convention will be contirued in this thesis and, where the possibility of ambiguity exist, a comment will be made to ensure clarity.

procedure, or a memory reference to the virtual address of some external data.

A. THE WALKTHROUGH

When the translator encounters an external reference in the source code of <Caller>, it will enter the symbolic name "Target|Entry_Name" in the symbolic name table for <Caller>. (The symbolic name table of <Caller> contains the symbolic names of external references and data associated with each entry point found in <Caller>. Additionally, the symbolic name table exists at run time.) The object code produced for the external reference to <Target|Entry_Name> (as found in <Caller>) consist of a procedure call to a virtual address in <Caller>'s linkage table[5]. (This virtual address is constructed at run time using a base register, called the linkage pointer, and some offset into Caller.link generated by the translator.) The virtual address called is an entry in Caller.link set aside for <Target|Entry_Name> and will be referred to as an outgoing link. The outgoing link has been initialized to invoke the linker and pass to the linker the offset (in Caller.sym) of the symbolic name "Target|Entry_Name". The linker uses this offset along with

---

[5] The symbolic name table of an object will be called object.sym, while object.link will refer to an object's linkage table. Thus <Caller>'s symbolic name table and linkage table become Caller.sym and Caller.link respectively.

the virtual address of the base of Caller.sym (which is stored in Caller.link), to access the symbolic name of the external reference. Once located, the linker will pass the symbolic name "Target" to the address space manager.

The address space manager first determines if an entry for <Target> already exist in the process reference table. If not, the address space manager will locate the object <Target> and have it assigned a segment number in the address space of the executing process. It will also make an entry for <Target> in the process reference table and return to the linker the segment number of <Target>. (It is at this point that <Target> is 'known' to the executing process.)

The linker now knows the segment number of <Target> and must create a linkage table for <Target> (if one has not already been constructed by an external reference to <Target> within another subroutine). A template accessable to the linker has been constructed by the translator for this purpose and is appended (after minor computations) to the end of the combined linkage table[6] (as Target.link). (The building of a linkage table for <Target> allows it to engage in dynamic linking.) Additionally, the starting address of Target.link is entered in a data structure known

---

[6] The combined linkage table contains the linkage tables of each object in a process. (Note that it is not necessary to utilize a combined linkage table in an implementation since each object's linkage table could be allocated its own segment.)

as the Linkage Address Table, making it available for future linking evaluations. (The linkage address table of a process can be considered an array containing the base address of each object's linkage table and is subscripted by the object's segment number.)

A complete virtual address for <Target|Entry_Name> can be constructed by searching Target.sym for "Entry_Name" to discover the entry point (offset) and incoming link offset associated with "Entry_Name". (An incoming link is a section of an object's linkage table set aside to allow the performance of housekeeping functions prior to invoking the object.)

The linker will now alter the outgoing link (in Caller.link) to jump to the incoming link (found in Target.link). The linker then constructs the incoming link to jump to the virtual address of <Target|Entry_Name> after setting the linkage pointer to point to Target.link. (The linkage pointer is a global pointer, e.g. hardware register, which always points to the currently executing subroutine's linkage table. Thus before execution in <Target> can commence, the linkage pointer must be set to point to Target.link. The reason for this will be discussed later.) After the outgoing and incoming links are executed, the process will be executing in <Target>.

When <Target> has finished it will execute a return

instruction. Recall that the only procedure call in the linkage sequence was <Caller>'s call to the outgoing link (in Caller.link) ensuring a return to <Caller> after the completion of <Target>. The final step is to reset the linkage pointer to the virtual (base) address of Caller.link. (This is done by the translated external reference in <Caller>.)

The steps followed for linking external data would be similar except data is not executed. Therefore, the outgoing link need not "invoke" the data (via the incoming link) but instead must allow <Caller> to reference the data. If indirect addressing is available, the outgoing link can be a storage location for the virtual address of the external data and can be referenced via an indirect addressing instruction. (Note that on the first reference, this indirect addressing instruction must be able to invoke the linker in some fashion. In Multics, this is done by generating a fault which invokes the linker as the fault handler.) If indirect addressing is not available (or cannot be used to invoke the linker on first reference to the data), the outgoing link can contain executable instructions which load some pointer with the virtual address of the data and then return the execution point to <Caller>.

B. A SYNOPSIS OF THE WALKTHROUGH

To provide the reader with an abreviated review of the steps to snap a link, the following synopsis is provided. Additionally, figure 1 is annotated with the number of each "step" to provide added clarity. When the executing procedure (i.e., <Caller>) encounters a translated external reference to <Target¦Entry_Name> for the first time, the following sequence of events transpires:

Step 1 - The execution point is transferred to the outgoing link (in Caller.link).

Step 2 - The linker is invoked by the initialized outgoing link. The linker is passed the offset of <Target¦Entry_Name>'s entry in Caller.sym as an argument.

Step 3 - The linker references Caller.sym and extracts the symbolic name "Target¦Entry_Name" and the offset (in Caller.link) of the (appropriate) outgoing link for <Target¦Entry_Name>.

Step 4 - The linker invokes the address space manager with the argument "Target".

Step 5 - The address space manager enters <Target> in the process address space (if necessary) and returns to the linker the segment number of <Target>.

Step 6 - The linker builds a linkage table for <Target> (not shown).

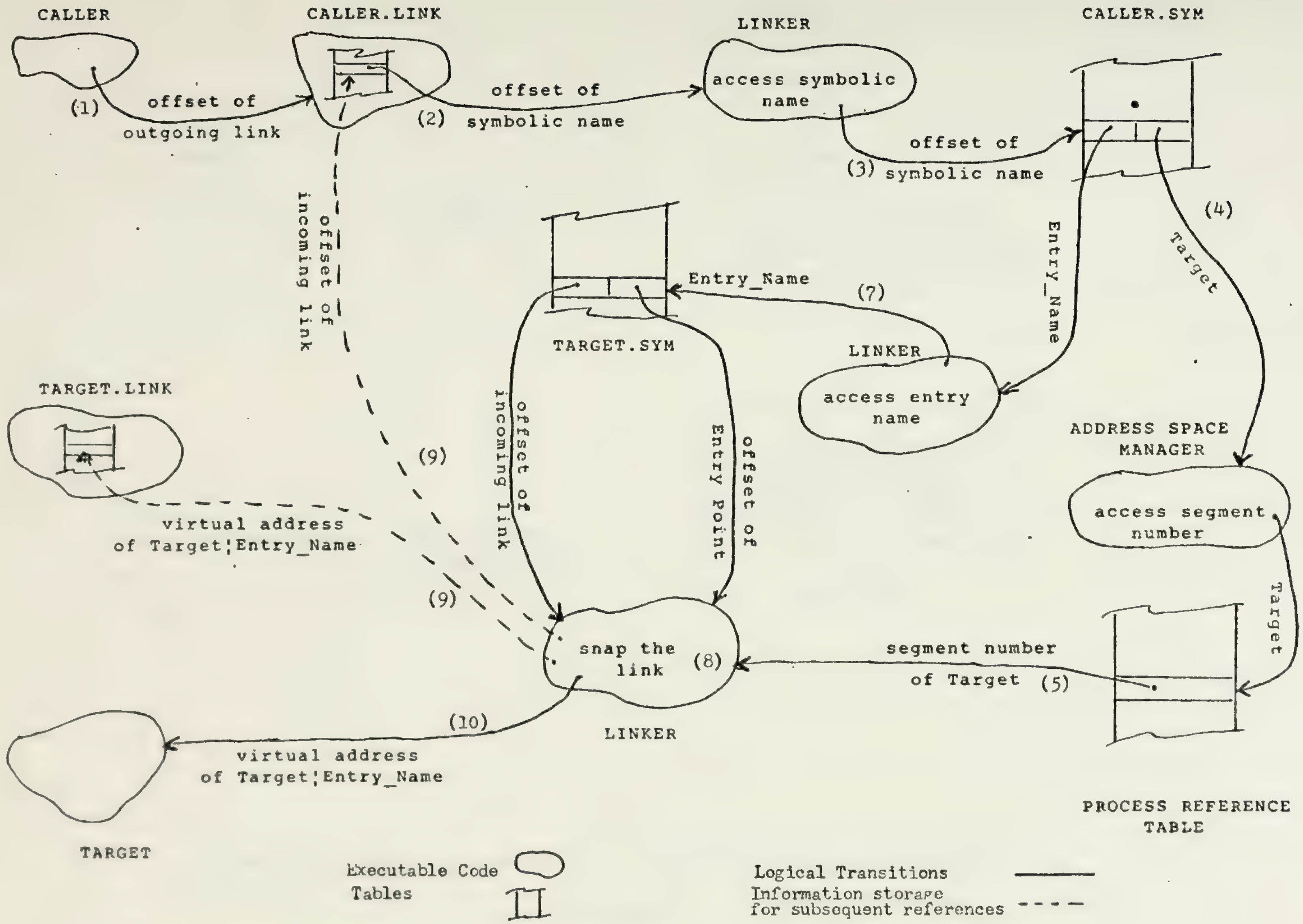
Step 7 - The linker searches Target.sym for "Entry_Name" and extracts the offset of the incoming link (for Entry_Name) in Target.link, and the entry_point associated with Entry_Name.

Step 8 - The linker computes the virtual address in <Target> associated with <Target¦Entry_Name> and the virtual address of Entry_Name's incoming link.

Step 9 - The linker establishes the link by entering a jump to the incoming link in the outgoing link (in Caller.link); and by loading the incoming link (in Target.link) with an instruction which loads the linkage pointer with the address of Target.link and a jump to the entry_point in <Target>.

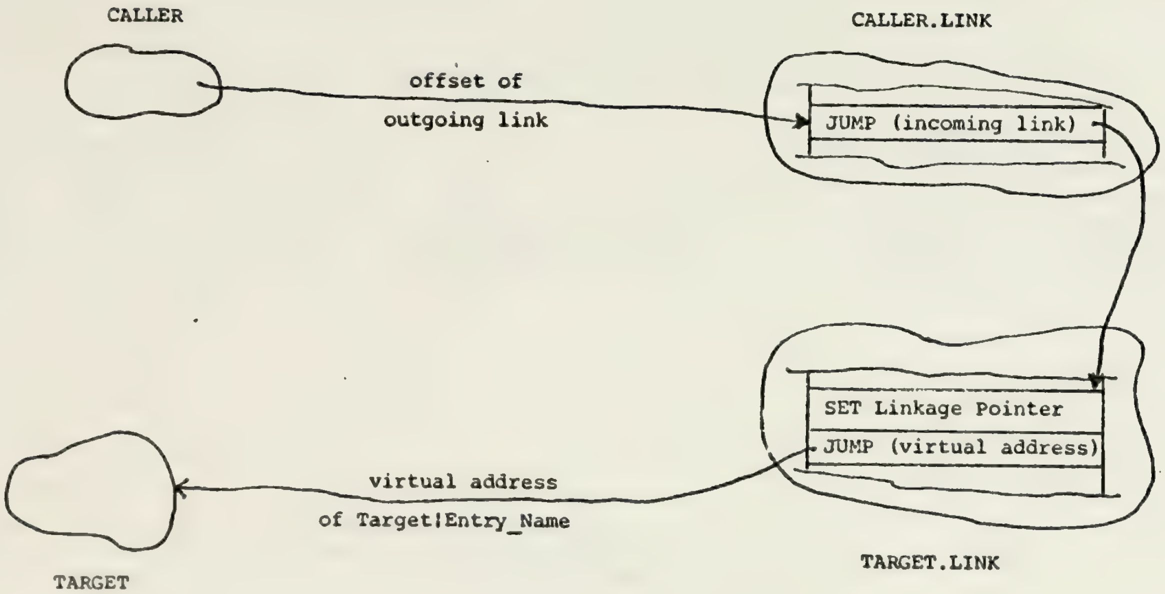
Step 10 - The linker invokes <Target> at the entry_point.

Figures 2 and 3 are included to show the execution sequence of a snapped link for procedures and data respectively. It is noted that a link that has already been established does not require the invocation of the linker but rather directly references the external object.



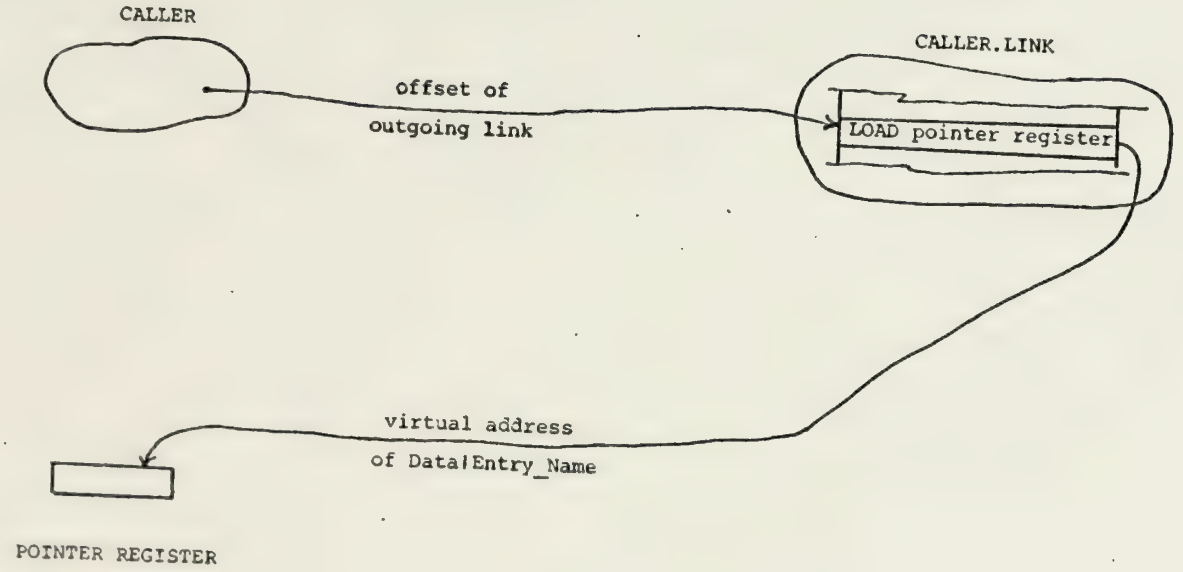
SEQUENCE OF EVENTS FOR SNAPPING A LINK TO THE PROCEDURE <TARGET¦ENTRY_NAME>

FIGURE 1



SEQUENCE OF EVENTS FOR SUBSEQUENT REFERENCES TO <TARGET|ENTRY_NAME>

FIGURE 2



SEQUENCE OF EVENTS FOR SUBSEQUENT REFERENCES TO <DATA|ENTRY_NAME>

FIGURE 3

IV. THE SPECIFICS OF DYNAMIC LINKING

A. FUNCTIONS OF A LINKER

Dynamic linking centers around the ability to alter impure code (linkage tables[7]) during run time. It is this feature which allows invocation of the linker on the first reference (to an object) and yet permits subsequent references to the same object to access that object directly (i.e., without invocation of the linker). Establishing, or snapping [11], a link does not represent all the functions desirable in a linker. Linkage tables must be constructed on the first reference (within a process) to an object, and system limitations may subsequently force the removal, or unlinking, of an object from a process address space.

1. Snapping a Link

a. Procedure Links

When snapping a link between procedures, the linker will initially be passed the offset (in Caller.sym) of (the entry for) the symbolic name "Target|Entry_Name". The linker can find Caller.sym via a pointer stored in Caller.link. (Recall that the linkage pointer always indicates the executing procedure's linkage table ensuring the linker can locate Caller.link.) Now the linker knows the

---

[7] It should be noted that linkage tables avoid the undesirable effects normally associated with impure code by being serially reusable and a per process entity (i.e., one linkage table per process for each object).

symbolic name of the object to be linked, but it must determine a virtual address within the object to be referenced.

In order to make <Target|Entry_Name> addressable, the linker must determine the segment number associated with <Target>, and the entry_point associated with Entry_Name. To determine the segment number of <Target>, the linker will invoke the address space manager passing the symbolic name "Target" as an argument. The address space manager will enter <Target> in the process address space (if it is not already) and return <Target>'s segment number to the linker Obtaining the segment number is trivial since the address space manager will return this information to the linker when passed the symbolic name "Target".

Finding the entry_point associated with Entry_Name requires access to Target.sym. As will be discussed, a second function of the linker is to construct a linkage table for <Target> (if one does not already exist as a result of some previous reference to <Target>). After Target.link has been constructed, to find Target.sym, the segment number of <Target> is first used to access (in the linkage address table) the virtual address of Target.link. (Recall that the linkage address table is an array of pointers to the linkage table of each object in a process

address space.) A pointer is found in Target.link to Target.sym.

It is proposed that, in an environment allowing multiple entry points into an object, each distinct entry name into an object be stored in the object's symbolic name table. In addition, the entry point (viz., the offset into the object) and the offset (in object.link) of the incoming link associated with each entry point will also be stored in object.sym. Thus, by searching Target.sym with the argument "Entry_Name", the linker can compute the entry_point and incoming link address necessary to snap a link to <Target¦Entry_Name>.

The first step in the actual snapping of the link is to alter the outgoing link (in Caller.link) from a jump to the linker to a jump to the incoming link (in Target.link). The address jumped to is formed by combining the segment number of Target.link (which is found in the linkage address table) with the offset (as stored in Target.sym) of the incoming link.

The second step is the building of the incoming link. The incoming link consist of two instructions. The first loads the linkage pointer (Lp) with the virtual address of Target.link ensuring that the linkage pointer always points to the currently executing procedure's linkage table. This is necessary to allow a procedure's translated

code (viz., object code segment) to reference an external object while remaining pure. A reference to an external object is achieved via the outgoing link; the virtual address of the outgoing link is computable at run time by adding a fixed (at translation time) offset to the linkage pointer and allowing the linkage pointer to vary during execution (see figure 4). Stated another way, it is the linkage pointer which allows (pure) translated code to jump to an entity (the outgoing link) which is not bound to a virtual address until run time.

The second instruction in the incoming link is a jump to the virtual address of <Target¦Entry_Name> (of the form <segment_number¦ entry_point>). Note that the incoming link may already exist in its snapped form as a result of some previous reference to <Target¦Entry_Name>. To identify this condition, the linker will first check a ´snapped link bit´ which is set if the incoming link is snapped. A snapped link is shown in figure 5.

One may observe that the outgoing and incoming links could be merged into one link consisting of a load linkage pointer instruction followed by a jump to <Target¦Entry_Name>. This change eliminates incoming links but effectively requires an ´incoming-type´ link to be constructed in each outgoing link referencing an object. This approach was not chosen since it requires the

SOURCE CODE

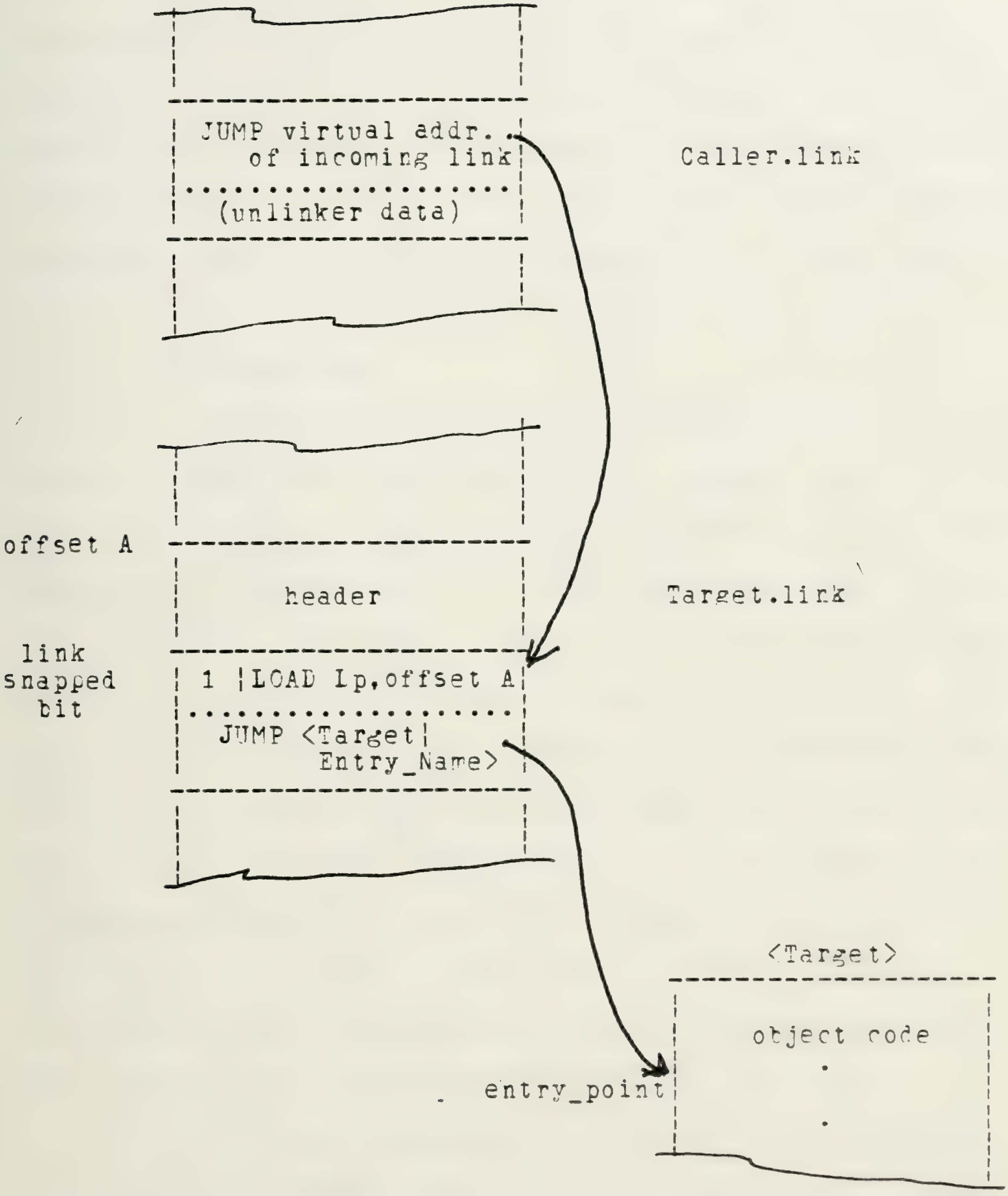
```
PROCEDURE EXAMPLE;
   DECLARE <Target> PROCEDURE EXTERNAL;

    /* code */

BEGIN   /* example */
            .
            .
   CALL  <Target|Entry_Name>;
            .
            .

END;   /* of example */
```

OBJECT CODE

```
/* begin example */
          .
          .
          .
CALL (Lp + offset of <Target|Entry_Name>'s outgoing link)
          .
          .
          .
/* end example */
```

TRANSLATED EXTERNAL REFERENCE

FIGURE 4



LINKAGE TABLE ENTRIES FOR A SNAPPED PROCEDURE LINK

FIGURE 5

construction of an 'incoming link' for each reference to a procedure (vice just one incoming link) and additionally results in multiple load linkage pointer instructions. (Note, however, that in either of these link formats, subsequent references to <Target¦Entry_Name> do not result in invocation of the linker.)

b. Data Links

For external data, the steps to snap a link are similar except the linker alters the outgoing link to an instruction which loads a pointer (preferably a register) with the virtual address of the external data, and a return instruction[8]. (As will be shown, it is necessary for data segments to have both linkage tables and symbolic name tables. This permits the linker to use essentially one algorithm to dynamically link both data and procedures.) Thus any subsequent references to the external data (<Data¦Entry_Name>) initiated by <Caller> would result in loading a pointer with the virtual address of <Data¦Entry_Name> followed by a return to <Caller> and would not result in additional linker calls.

As with procedures, it is desirable to reference multiple, symbolically named locations (viz., entry points)

---

[8] As has been previously discussed, it is necessary to use this form of outgoing link if the processor hardware cannot support an indirect addressing instruction to invoke the linker (on first reference) and subsequently access the virtual address of the external data.

in a data structure. This implies that <Data> must undergo a translation to identify entry names and entry points and furthermore, must have a symbolic name table in which this information is stored. It is also necessary, given this condition, that <Data> have a simplified linkage table consisting of a linkage table header. (The contents of a linkage table header will be presented later.)

c. Construction of a List of Snapped Links

For each object in a process address space, it may be desirable for the linker to construct a linked list which contains a pointer to each snapped outgoing link referencing that object. This linked list is basically used to provide a record of references to an object to permit unlinking an object from an address space. (Unlinking will be discussed in more detail later.) A pointer to the start of this linked list would be stored in the header of the object's linkage table and new entries to the linked list would be entered at the head of the list (when snapping an outgoing link). The linked list could easily be implemented by storing a list pointer in each snapped outgoing link.

2. Building Linkage Tables

Before <Target> can commence execution, it must have a linkage table in which snapped links can be stored. This allows <Target> to engage in dynamic linking (if it is a procedure). There exist two circumstances under which the

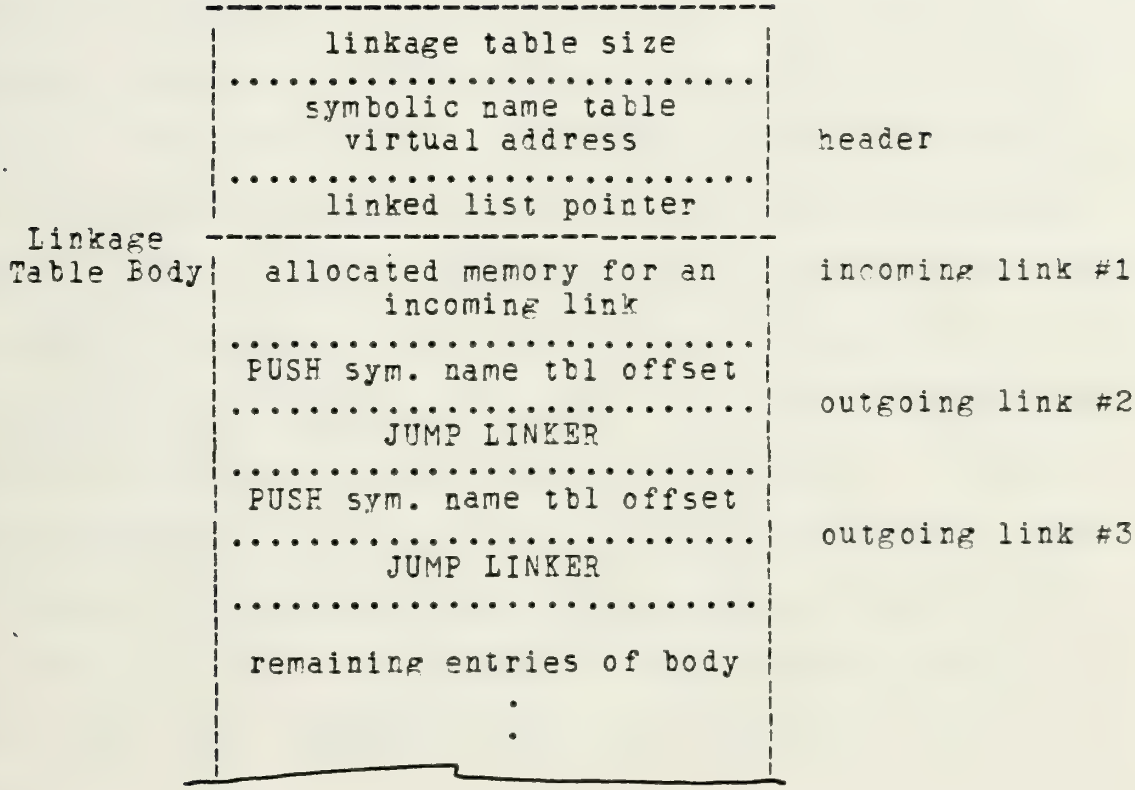
linkage table must be built. The first, and obvious, situation is when an external object is dynamically linked to a process. The second is when a program is initially started executing (viz., during process initialization). However the steps involved in these two cases do not differ, allowing the same module of the linker to be utilized in both instances.

To build a linkage table, the linker will access a template for an external object (or program) that was constructed during translation. The template is an exact duplicate of object.link with the exception of the symbolic name table virtual address. The linker must therefore only add the segment number of <Target> to the symbolic name table offset as found in the template to obtain a complete virtual address (for Target.sym). (This approach assumes Target.sym is a part of the translated code of <Target>.) The remainder of the template is then appended to the combined linkage table[9]. An example of an initialized linkage table (and thus a template) is given in figure 6.

There are two problems related to the implementation of this linker function which require discussion. The first

---

[9] It is not necessary for an implementation to include the combined linkage table since individual linkage tables can be assigned unique segment numbers. In fact, in a multidomain environment [8], it is desired to assign linkage tables to separate segments since this permits the dynamic linker to be domain independent (in accordance with the design criteria of Table 1).


linkage table size
symbolic name table virtual address
header
linked list pointer
Linkage Table Body
allocated memory for an incoming link
incoming link #1
PUSH sym. name tbl offset
outgoing link #2
JUMP LINKER
PUSH sym. name tbl offset
outgoing link #3
JUMP LINKER
remaining entries of body

INITIALIZED LINKAGE TABLE

FIGURE 6

question involves where the template is located in a process address space. One does not, in general, want the template to be a part of the object code since this will result in an entity (the template) which is used only once becoming an extraneous part of a process. (Note that system limitations may force this shortcoming on an implementation.) A solution in a non-segmented system is to make the template a separate file. (One may not wish to do this in a segmented system if the number of segments represents a limited asset and a file corresponds to a segment since this would require assigning the template its own segment number.) However, in a demand paging environment, the template can be a part of the object code since it will only reside in memory when required and will then be 'paged' out. Because it will never again be referenced, the template will never again be loaded into memory.

This leads to the second problem of ensuring the linker can find the template when it is a part of the object code. There are several solutions to this, the most simple of which is to place a pointer to the template at some known location in the object code. Another solution would entail making the template a separate file. Thus when building a linkage table, the template is brought into a process address space, copied into the combined linkage table, and then deleted from the process address space.

3. Unsnapping Objects

It may be necessary to remove an object from the address space of a program. This situation may occur, for example, when using the Z8000 processor [12] with one memory management unit (MMU). Since this hardware configuration allows a maximum of 64 segments (some of which will be allocated to the operating system), it is entirely possible that a process may require in excess of the maximum number of available segments. It is desirable then to be able to remove an object from the process address space and unsnap all outgoing links referring to that object.

The unsnapping of an outgoing link is a simple procedure. The snapped outgoing link is merely replaced by an entry equivalent to the original, unsnapped outgoing link. More specifically, this unsnapped link consist of code to pass the linker the offset of the external object's

symbolic name in object.sym followed by the invocation of the linker. (For simplicity, it will be assumed that the linker is invoked via a jump instruction.) This implies that a portion of each snapped link must be set aside to store the offset of the symbolic name for use during unlinking[10].

The first step in the unlinking process occurs when the address space manager, after being requested by the linker to add an object to a process address space, returns a message to the linker indicating no segment numbers are available (if this is the case). The linker would then cause a segment to be deallocated.

If desired, the object's linkage table (object.link), can be deleted from the combined linkage table by performing a compaction on the combined linkage table. (Note that compaction is not necessary since, aside from resulting in unused memory in the combined linkage table, if the deleted segment is reentered in the process address space, a new linkage table will be built and appended to the combined linkage table.) If a compaction is

---

[10] Note that all information necessary to reset the link (thus deleting the requirement to store the offset in the linkage table) is available in the combined linkage table, the subroutine offset table and the template. However, the steps necessary to extract this data are rather involved and the alternative of saving the offset within a snapped link is suggested unless infrequent unlinking evolutions are expected.

done, the deleted linkage table contains threads in the linked list of other segments, which must be removed without destroying the linked list they were a part of. One solution to this problem is to implement a doubly linked or circular linked list (by having the last entry of the list point to the linkage address table instead of being set to nil). Now, prior to removing object.link, the linker could find and adjust each thread (of a linked list) with a node in object.link ensuring the integrity of other segments' linked list.

Compaction presents two other problems. First, when object.link is removed, other subroutines' linkage tables may be relocated within the combined linkage table thus receiving new virtual addresses. This requires that the linkage address table values for those linkage tables along with linked list threads pointing into them to be adjusted accordingly. The correction must be done prior to actually compacting (because linked list threads in the deleted linkage table will be lost during compaction) and requires that addresses in the combined linkage table (i.e., subroutine offset table, linked list, and snapped link addresses) be corrected by the size of the removed linkage table. A second problem relates to snapped outgoing links which jump to incoming links in relocated linkage. These must also be adjusted by the size of object.link. Note that

a subroutine's linked list identifies each outgoing link that jumps into its linkage table. Therefore, every procedure segment whose linkage address table value requires correcting must have each entry in its linked list updated.

When unsnapping, the linked list (constructed by the linker) is traversed and each entry in the list is reinitialized. Note that unlinking affects many subroutine linkage tables yet the linkage pointer still points to object.link for the subroutine which originally invoked the linker. This implies that linked list pointers must either be complete virtual addresses or relative to the start of the combined linkage table (i.e., they cannot be relative to the linkage pointer.)

An alternative to a linked list implementation is to have the linker search the combined linkage table for all snapped outgoing links referencing the deleted segment and reset each one found. (This is a less general solution since it requires the linker to know the format of all possible linkage table entries in order to identify those which must be reset.) Once all linkage table entries have been reinitialized, the object's linkage address table entry is set to nil, and the object and its linkage table (if desired) are removed from the process address space.

B. OPERATING SYSTEM SUPPORT

1. The Address Space Manager

As has been noted, before a link to an object can be snapped, the object must first be entered in the address space of a process. A request to enter an object (i.e., make it known) is forwarded from the linker to the address space manager. The address space manager will be passed the symbolic name of the object that is to be made accessable and will first search each entry in the process reference table to determine if an entry already exist for the object. If so, it will return to the linker the segment number of the object.

If the object is not accessable, the address space manager must first call on File Management to locate the object. After the object is located, Memory Management is invoked to assign a segment number to the object[11]. If Memory Management were to indicate that it had no segment numbers left to assign, the address space manager would return to the linker a message to this effect.

---

[11] It is realized that this represents a very vague description of how an object is located and assigned a segment number. However, since the exact steps involved are highly dependent on the operating environment and are fundamental to most multiprogramming systems, it is felt that adequate information exist elsewhere to allow implementation of these functions without discussing them in this thesis. Note that the file system in use may be extremely sophisticated as in Multics [11], or represent a simple one-to-one mapping of symbolic names to corresponding files.

2. The Process Reference Table

The process reference table contains an entry for each object in the address space of a process. The format for an entry (figure 7) includes the symbolic name of the object along with the segment number of the object. A third item which may be found in the process reference table is a removal status reflecting the priority of an object for removal when unlinking.

Note that unlike a symbolic name table entry, the symbolic name found in the process reference table does not include entry names. For example, a process may contain external references to <Target|Entry_Name_1> and <Target|Entry_Name_2>, but the process reference table would only contain one entry for <Target>.

symbolic name	segment number	removal priority

Figure 7 - Process Reference Table Entry

3. Object Deletion from a Process Address Space

In conjunction with the linker, a module of the operating system must exist to delete an object from the address space of a process. When invoked by the linker, this module would use some policy, such as least recently used or

first-in, first-out, to select an object for removal. The module would notify Memory Management that the object's segment number is no longer in use and reset the object's entry in the process reference table to nil. The module would then inform the linker of the segment number of the deleted object. The linker can now unsnap links to the object.

It is useful to point out policy considerations for selecting an object for removal. To begin, note that each time a link is snapped to an object, the address space manager is called to look up the segment number of the referenced object. It may, therefore, be advantageous to keep track of the number of links to an object to avoid removal of a segment which is referenced many times. (One should not, however, strictly delete the object referenced the least number of times since this may well be the last object entered in the address space and, applying the principal of locality, be subject to further use in the near future.)

Another important item to be considered before selecting a subroutine for removal is whether it will eventually be returned to by the currently executing procedure (i.e., it has a current activation record). As an example, say procedure A called procedure B which called procedure C. But before C could be linked an unlinking

evolution was required. Certainly one would not want to remove A or B to make virtual memory available for C since these two procedures would be returned to when C completed executing and the linking process has only been defined during a procedure call. Thus, if A or B were unlinked, C would return to a non-existant module which it could not link to or access (since A or B would no longer be in the process address space.)

If the information necessary to determine whether a procedure has a current activation record is not readily available, there is an easily implementable mechanism for determining this. A counter can be assigned to each procedure (in a process address space) that would be incremented or decremented as the procedure is invoked or completes execution. Thus, a procedure whose counter is zero has no current activation records and is available for removal. The counter could be updated by code in the snapped link and could be located in a procedure's linkage table or linkage address table entry. This implies that the linker must be involved in the selection of an object for removal.

4. Process Initialization

Process initialization involves those functions which must be carried out by the operating system prior to commencement of program execution. A brief review of these functions is offered at this time with a more detailed

discussion available in work by Janson [7, 8].

Before a process can commence execution of a program, the program's linkage table (program.link) and linkage address table must be allocated a section of the process address space and both tables must be initialized (or built from a template in the case of program.link). Additionally the linkage pointer must be set to point to program.link. The operating system must initialize the process reference table with the applicable data for the program to be executed. Once this is accomplished, calls by <program> can be dynamically linked.

C. TRANSLATOR SUPPORT

The process of dynamic linking is only practical if the translator, whether a compiler or assembler, has been designed to support dynamic linking. In a (translator) supported system, the translator must be able to identify external references, build the symbolic name table and linkage table template, and identify entry points and entry names. A translator will be assumed to produce relocatable object code allowing dynamic relocation of object code segments--either by relocation hardware or software.

Together, the translator and the linker must meet two requirements. First, the object code must remain pure during the linking process to allow use of shared procedure segments in a multiprogramming environment (i.e., the pure

object code criterion of Table 1). In addition, the code produced by the translator along with the steps followed in the linking process must not limit features of the source language (i.e., the syntactic compatability criterion).

1. External References

A translator must be able to identify external references and convert them into object code which will result in a call to the outgoing link (figure 4). The call produced by the translator is to an address which can be expressed as the value of the linkage pointer plus some offset. Since the translator constructs the linkage table template, it knows the relative offset for a symbolic name's outgoing link in the linkage table. As has been noted, because the linkage pointer identifies the beginning of the executing procedure's linkage table, the object code for an external reference can be designed to call the outgoing link desired. (The use of the linkage pointer ensures the purity of a procedure's translated code.)

2. Symbolic Name Tables and Templates

The translator builds both the symbolic name table and the linkage table template. This should not present any major problem for the translator since all information required to construct these two items is, in general, either easily computable or found in the translator's symbol table. Because the translator builds both, it is not necessary for

entries in either to be of uniform size. The translator, for example, knows the offset (i.e., starting location) of a symbolic name table entry. Therefore, when the translator constructs the linkage table template, each outgoing link can be initialized to pass this offset to the linker (on first reference of an object.)

Notice that a one-to-one correspondence exist between entries in the linkage table body and the symbolic name table. Thus, if the symbolic name table is constructed first, the construction of the template becomes trivial. After the header of the template is built, the symbolic name table is scanned and an outgoing or incoming link is initialized within the template (depending on the type of symbolic name encountered). After each template entry is constructed, the offset of the link from the start of the template can be stored into its respective entry in object.sym.

3. Entry_Names and Entry_Points

The translator should be able to recognize both entry names and their associated entry points and make appropriate linkage table and symbolic name table entries accordingly. The inclusion of entry points in the implementation of a dynamic linker is highly desirable, particularly in a system with a limited virtual memory size. In this environment the number of unlinking evolutions may

be significantly reduced by using entry points to combine small data or procedure objects into larger ones without losing the smaller object's addressability[12].

---

[12] This process is known as binding in Multics [11].

D. DYNAMIC LINKING TABLES

The following is a discussion of the various tables associated with a dynamic linker. The formats presented do not represent the only structures possible; however, they contain all information necessary for dynamic linking.

1. The Symbolic Name Table

An entry in the symbolic name table (figure 8) in addition to the symbolic name includes two other items. The first is a descriptor consisting of a type bit to identify the object as procedure or data; an identity bit to classify the symbolic name as an external reference verses entry name; and a size field to pass to the linker the number of characters in the symbolic name.

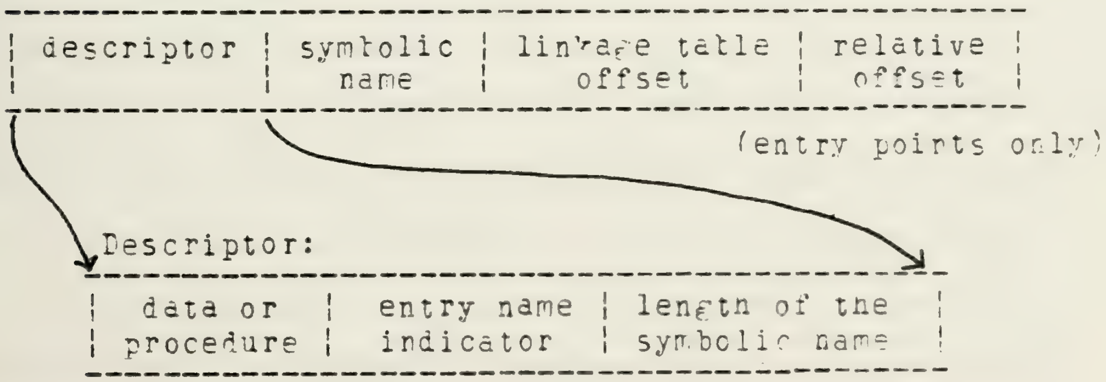


Figure 8 - Symbolic Name Table Entry.

A second item to be included is the offset of a symbolic name's entry in the subroutine's linkage table. For external references, the inclusion of the offset in a symbolic name table entry is not necessary; however, its inclusion does remove the requirement for the linker to save this information when it (the linker) is invoked by an outgoing link. However, for an access (entry point) into an object, the offset (of the incoming link) must be included in the symbolic name table to ensure the linker knows where, within object.link, to construct the incoming link. The third item found in the symbolic name table is the entry point (offset) associated with each entry name declared within an object. (The entry point is used to construct a virtual address of the form <segment_number|entry_point>. This virtual address is used in the incoming link to invoke the called external procedure.)

It may be desirable to separate the symbolic name table into two sections consisting of external references and entry points. Assuming the entry points follow the external references, a pointer to the beginning of the entry points should be stored at the beginning of the table to allow the dynamic linker to jump directly to the entry point section when required. This feature would permit faster access for both since each would be stored in a smaller data structure. If this table organization is used it would not

be necessary to include an identity bit in the descriptor of an entry. (Note that the symbolic name table is searched for entry points only, since external references are accesses directly via the outgoing link.)

It is natural to ask where the symbolic name table of an object is located within a process. It is suggested that for procedures, the symbolic name table be appended to the end of a procedure's object code. This will require only one copy of the symbolic name table (which represents a pure data structure) in a shared, multiprogramming environment. However, for external data, the symbolic name table cannot be located at the end of the data since it will limit the ability of the data structure to grow dynamically. A better solution would be to merge the data.sym with data.link and store the two in the combined linkage table. This format allows the data to be based at offset zero and grow dynamically. (The general form of a data symbolic name/linkage table is given in figure 9.)

2. The Linkage Table

a. The Initialized Linkage Table

The initialized linkage table is shown in figure 6. The header of the linkage table contains three items. The first is the size of the linkage table. This item tells the linker the size of the template when building an object's linkage table and also is used by the linker to adjust

```
Linkage   -------------------------------
 Table    |  linkage table/symbolic      |
          |     name table size          |
          |..............................|
          |     linked list pointer      |
Symbolic  -------------------------------
  Name    | des-    |  Entry_  |  Entry_  |
 Table    |criptor  |  Name_1  | Point_1  |
          |..............................|
          | des-    |  Entry_  |  Entry_  |
          |criptor  |  Name_2  | Point_2  |
          |..............................|
          |                              |
          |                              |
          |        remainder of          |
          |     symbolic name table      |
          |                              |
```

DATA SEGMENT SYMBOLIC NAME TABLE
AND LINKAGE TABLE

FIGURE 9

linkage table addresses when removing a linkage table (during unlinking). (Recall that linked list threads, linkage address table entries, and jumps within the linkage table must be adjusted by the size of a removed linkage table during compaction of the combined linkage table.) The second and third items found in the header consist of the virtual address of the symbolic name table and a pointer to the head (i.e., a snapped outgoing link to the object) of the linked list used in unlinking.

Each outgoing link in the body of the linkage table template is initialized to two instructions. The first instruction passes the entry's offset in the symbolic name table to the linker (as an argument). The second is an instruction which results in the invocation of the linker. Logically, the two instructions found in the initialized outgoing link equate to:

```
   CALL Linker (symbolic_name_table_offset)
```

The designer can chose from three basic mechanisms that may be used to invoke the linker. First, if the translator knows the virtual address of the linker (such as a fixed or reserved segment number), then the outgoing links in a template can be tailored to invoke the linker directly (e.g., JUMP virtual address of <linker>). The second method is to invoke the linker by a hardware fault which will result in the linker being called as the fault

handler. The translator would therefore, initialize each outgoing link to push the offset of the symbolic name on the machine stack and then induce a hardware fault. The third mechanism is for the linker to enter its own virtual address in each outgoing link as it builds a procedure´s linkage table. (This represents the least desirable technique since it requires the linker to know the format of the body of a template and furthermore is much slower since the template must be scanned as the linkage table is built.)

b. Format of Snapped Links

A format for snapped outgoing links to external data and procedures are shown in figure 10. The snapped outgoing link for a procedure consist of a jump to the incoming link in the called procedure´s link for an external procedure´s linkage table. The snapped incoming link loads the linkage pointer with the virtual address of the called procedure´s linkage table (viz., Target.link), and then jumps to the called procedure (as defined by some entry point). For external data, the snapped outgoing link consist of an instruction which loads a register with the virtual address of the data followed by a return instruction. (Recall that this technique is used when the available hardware does not support an indirect addressing approach.)

The two items common to both entries (as shown in figure 10), ´offset´ and ´linked list pointer´, represent

```
              ------------------------------------
Data          | LOAD ptr, address of data        |
              |..................................|
Entry         |             RETURN               |
              |..................................|
              |offset |linked list pointer       |
              ------------------------------------


              ------------------------------------
Procedure     |  JUMP to virtual address         |
              |..................................|
Entry         |offset |linked list pointer       |
              ------------------------------------
```

DATA AND PROCEDURE SNAPPED OUTGOING LINKS

FIGURE 12

information to be used during unlinking. The offset (of the symbolic name table entry corresponding to the outgoing link) is used when resetting the entry to its initialized form while the linked list pointer allows the unlinker to find each entry in the combined linkage table which references the object being removed.

3. The Linkage Address Table

To facilitate each access to a subroutine's linkage table within the combined linkage table, the linkage address table is used. Entries are subscripted by segment number and contain the offset of an object's linkage table within the combined linkage table. (Note that if linkage tables were allocated individual segments, vice a portion of the combined linkage table, the linkage address table would contain the virtual address of an object's linkage table.)

The problem arises as to where in a process address space the linkage address table should be located. One would like to avoid allocating the linkage address table its own segment and pointer register since these resources within a microprocessor are usually limited. Assuming the linkage address table is initialized at process creation and is a fixed length, a possible solution is to place it at the head of the combined linkage table. If this approach is used, the table's base address would be the segment number of the linkage table (which is stored in the linkage pointer) with

an offset of zero.

F. IMPLEMENTATION OF ENTRY_NAMES AND ENTRY_POINTS

To avoid confusion, some of the fine points related to the implementation of entry names and entry points will be discussed at this time.

First, if an object has multiple entry points declared within it, each entry point must have a unique entry in object.sym and a unique incoming link in object.link. This is logical since each entry point defines a distinct location in an object. Secondly, if a procedure contains external references to several entry points within the same object, each unique reference must have its own entry within the procedure's symbolic name table and its own outgoing link. (For example, <Target¦Entry_Name_1> and <Target¦Entry_Name_2> represent distinct references.)

Notice that the start of an object represents an (frequently implied) entry point which must be included in the object's symbolic name table and have an incoming link. However, one would like not to explicitly include such an 'entry point' (e.g. <Target¦Target>) in an external reference. Therefore, it is suggested that an implementation default to this implied entry point in the absence of an entry name.

V. LINKING WITHOUT TRANSLATOR SUPPORT

In all probability, the initial implementation of a dynamic linker will not enjoy the translator support which has previously been assumed to exist. Yet, within reasonable limitations, one would like to be able to utilizing the features of dynamic linking in an unsupported[13] environment. Furthermore, it is desirable to be able to use one dynamic linker for both supported and unsupported procedures, to be able to execute both supported and unsupported modules within a process, and to be able to call an external procedure from a supported module without having to specifically declare the procedure to be supported or unsupported. (This implies the linker must be able to differentiate supported procedures from unsupported ones.) An implementation is proposed which achieves these goal.

A. THE INTERFACE MODULES

In an unsupported subroutine, the linker should be invoked via a data or procedure interface module. Two separate modules are suggested since, besides the fact that their functions differ, the data interface module must return the virtual address of the external data to the point

---

[13] For the purposes of this thesis, 'supported' will be used when referring to an environment in which the translator supports dynamic linking while 'unsupported' will be used to reference environments which lack this feature.

of call while the procedure interface module merely executes the snapped link [14]. Conceptually, an interface module carries out those functions which, in a supported system, require some translator support. These functions include building the symbolic name table and the linkage table, invoking the linker, and executing a snapped link.

1. Linking of Procedures

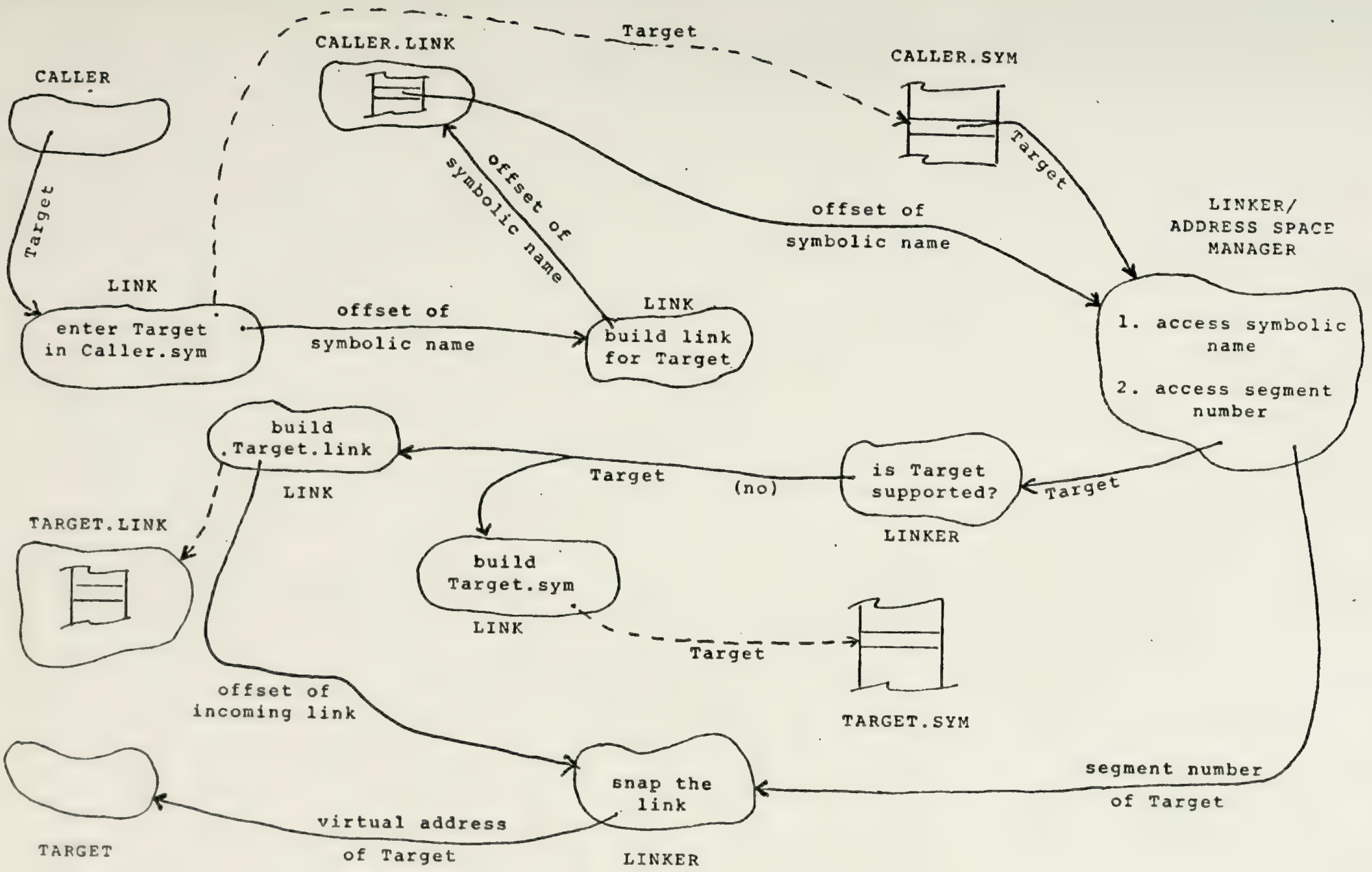
To best describe the functions of the procedure interface module, the steps to dynamically link the unsupported procedure <Target> to the unsupported procedure <Caller> will be traced (figure 11). It will be assumed that the procedure interface module is called as follows:

```
CALL LINK$PROC(Target, parameter_1, parameter_2, . . . )
```

The first function that <LINK$PROC> would perform would be to save the value of the interface linkage pointer on a software stack. Because a translator which does not support dynamic linking would not know that the linkage pointer register is not available for general use, in all probability object code produced would utilize the linkage pointer register requiring an interface linkage pointer be established and saved in software. (In a supported system

---

[14] This requires that two interface modules (one for procedures and one for data) be implemented due to the fact that most higher level languages have a different syntax for procedures which return arguments to the point of call (verses those which do not).



SEQUENCE OF EVENTS FOR LINKING UNSUPPORTED OBJECTS

FIGURE 11

saving the linkage pointer register is accomplished by the translated code.) This implies that there are two linkage pointers (viz., a hardware linkage pointer and an interface linkage pointer), and both must be initialized to point to the beginning of the linkage table for <program> at process initialization. It should be noted that the last instruction of <LINK$PROC> must reset the interface linkage pointer by poping the saved value off a software stack prior to returning to <Caller>.

<LINK$PROC> will then check to see if an entry for <Target> exist in Caller.sym. If not, <LINK$PROC> will enter <Target> in Caller.sym along with the offset of the outgoing link for <Target> in Caller.link. <LINK$PROC> is able to enter this offset because it has constructed an outgoing link for <Target> in the next free location in Caller.link. (The outgoing link for <Target> is of the same format found in a supported system linkage table.) This outgoing link is executed and the linker is invoked. (These steps ensure the use of the same linker for both supported and unsupported linking since the method of linker invocation does not change.)

It should be pointed out that had an entry for <Target> already existed, then <Target> would have already been linked to <Caller> and <LINK$PROC> would only have to execute the snapped link (which it can find since the offset

of the snapped incoming link in Caller.link is saved in Caller.sym.

Once the linker is called, it will first determine if <Target> is a supported or unsupported procedure. The actual mechanism used to perform this check will vary depending on the operating system. One means of performing this check is to tag modules within the file system. An alternative would be to tailor the first byte of a supported module to identify it as such. (One must ensure when using this method that an unsupported module cannot have the same bit pattern for its first byte.) In this thesis it will be assumed that the linker can query the file system to determine whether a module (external object) is supported or not. The ability of the linker to accomplish this check allows an external reference within a supported subroutine to have the same format regardless of whether the external object referenced is supported or not. This prevents having to modify and retranslate modules when an unsupported object is retranslated in a supported environment.

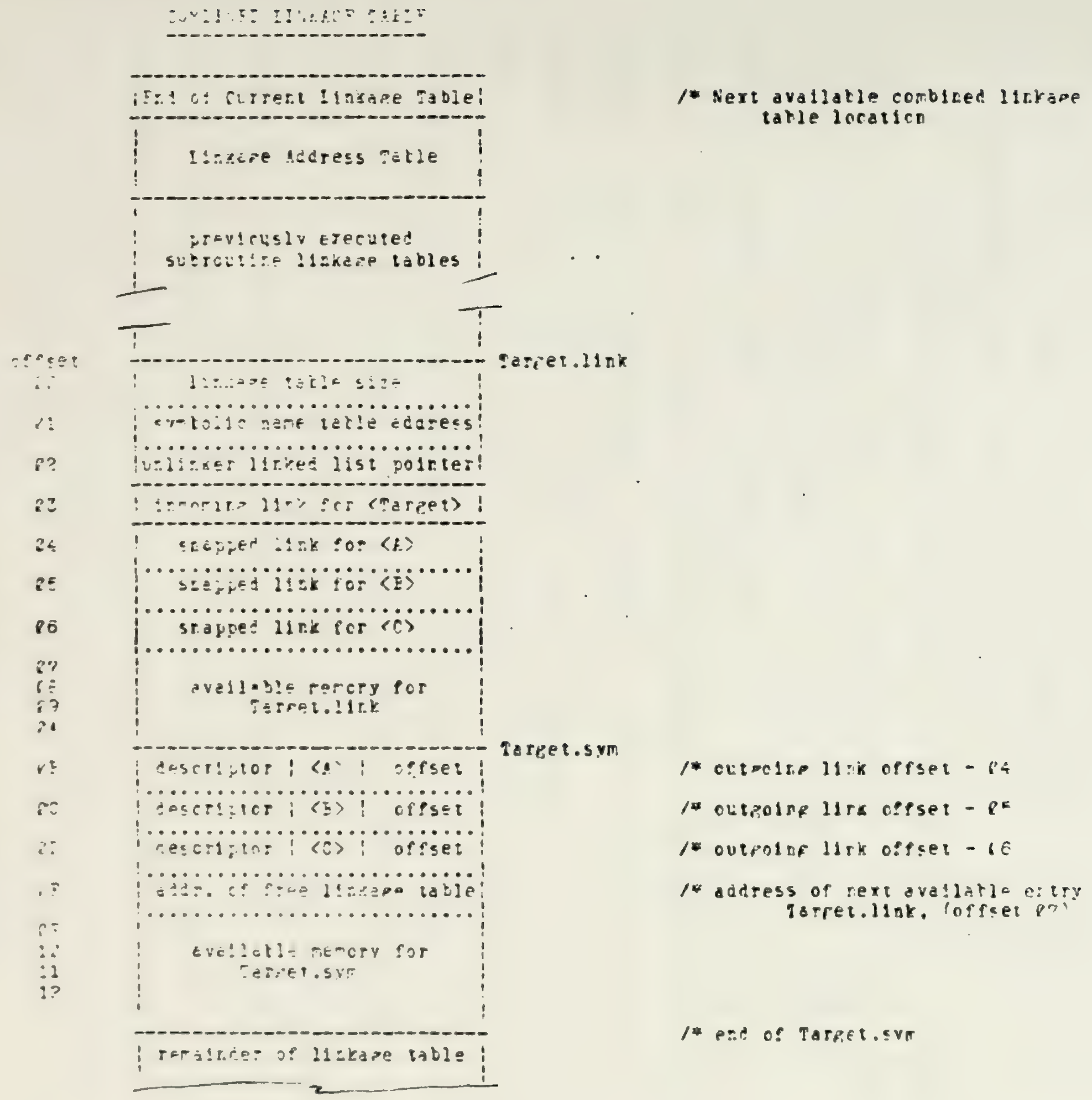
When the linker determines that <Target> is unsupported, it will call on a routine in <LINK$PROC> to allocate a section of the combined linkage table to be used as Target.sym and Target.link. This implies that the next free location in the combined linkage table must be available to <LINK$PROC> in addition to the linker (for

constructing linkage tables since <LINK$PROC> must build the linkage table for an unsupported subroutine. Target.sym can be located within Target.link (vice Target's object code) since the linker finds Target.sym via a pointer in Target.link (figure 12). Additionally, <LINK$PROC> will construct <Target>'s linkage address table entry and will initialize the header of Target.link.

The linker needs to know whether <Target> is supported or not for one other important reason. Execution of <Target> is initiated via a jump from <Caller>.link to an incoming link in Target.link. The incoming link normally consist of an instruction to set the linkage pointer register to point to Target.link followed by a jump to <Target>. However, if <Target> is unsupported, it is the interface linkage pointer vice the linkage pointer register which must be set requiring the linker be able to distinguish between supported and unsupported external procedures and snap incoming links accordingly. Thus the unsupported incoming link will be of the form:

```
Interface Linkage pointer = Base address of Target.link
Jump to <Target>
```

Note that <LINK$PROC> is passed not only the symbolic name "Target", but also all of <Target>'s parameters. This implies that <LINK$PROC> must be able to pass these parameters to <Target> in accordance with the


COMBINED LINKAGE TABLE
End of Current Linkage Table
/* Next available combined linkage table location
Linkage Address Table
previously executed subroutine linkage tables
offset
Target.link
00
linkage table size
01
symbolic name table address
02
unlinker linked list pointer
03
incoming link for <Target>
04
snapped link for <A>
05
snapped link for <B>
06
snapped link for <C>
07
08
09
0A
available memory for Target.link
Target.sym
0B
descriptor | <A> | offset
/* outgoing link offset - 04
0C
descriptor | <B> | offset
/* outgoing link offset - 05
0D
descriptor | <C> | offset
/* outgoing link offset - 06
0E
addr. of free linkage table
/* address of next available entry Target.link, (offset 07)
0F
10
11
12
available memory for Target.sym
/* end of Target.sym
remainder of linkage table

UNSUPPORTED LINKAGE TABLE

FIGURE 12

conventions of the translator which compiled <Target>.

2. Linking of Data

The sequence of events to link (the unsupported object) <Data> to <Caller> would be quite similar to those linking <Target>. Assuming <Data> had not yet been referenced by the executing process, the interface module <LINK$DATA> would build an outgoing link for <Data> in Caller.link and enter the symbolic name "Data" in Caller.sym (as <LINK$PROC> does for <Target>).

The linker would then be invoked and, upon determining <Data> to be unsupported, would call <LINK$DATA>. <LINK$DATA> would construct a linkage table for <Data>; however, the construction of Data.link would be trivial since it consist of only a linked list pointer and a linkage table size/entry (figure 8). As will be discussed, unsupported objects cannot have multiple entry points; therefore, <Data> does not require a symbolic name table. Following the construction of Data.link, the linker would snap the link between <Data> and <Caller>. The snapped link in this situation would differ somewhat in that once snapped, the link will be used by <LINK$DATA> to obtain the virtual address of <Data>. <LINK$DATA> completes the reference to <Data> by returning <Data>'s virtual address to (the point of call) in <Caller>.

B. LIMITATIONS OF UNSUPPORTED LINKING

There are four major disadvantages when linking in an unsupported environment. Three of these represent violations of design criteria (as specified in Table 1) while the fourth, the inability to implement multiple entry points, is considered a limitation in the flexibility associated with a dynamic linking environment.

The first disadvantage is that unsupported linking results in excessive overhead for subsequent references to an external object, as required by the limited overhead criterion. This is a direct result of the fact that the interface module must be invoked for each external reference to perform those bookkeeping functions (such as manipulating the interface linkage pointer) which in a supported environment are performed by the translated external reference and the snapped link.

A second disadvantage is that an external procedure must be linked before it can be passed to a subroutine as a parameter. This contradicts the delayed binding criterion. Furthermore, to pass an external procedure as a parameter requires a third interface module. A third interface module is called for since <LINK$PROC> can only link and invoke external procedures whereas to pass a procedure as a parameter, it is necessary to have access to the procedure´s virtual address. (In the case of an external procedure, it

is sufficient to pass the virtual address of the procedure's outgoing link vice the virtual address of the external procedure itself.) Therefore, the third interface module will snap the link (in violation of the delayed binding criterion) and return the virtual address of the external procedure's outgoing link to the point of call.

The third disadvantage involves a violation of the syntactic compatability criterion for external data. Note that the utilization of external data is limited to a (PL/1 or PL/M) based variable structure since <LINK$DATA> can only return the virtual address (of the external data) to the point of call.

The final disadvantage is that multiple entry points cannot be implemented in an unsupported object. Since the (translator constructed) symbolic name table is necessary to retain the entry name and entry point associated with an access into an object, an unsupported object can only be referenced at its conventional starting location.

VI. HARDWARE TO ENHANCE DYNAMIC LINKING

Even though care has been taken to develop a dynamic linker which is not dependent on the availability of certain hardware features, there are hardware capabilities which are desirable in a dynamic linking environment. In general, these features can be divided into two general categories: those which effect the design of the linker; and those which impact on system performance.

It is emphasized that the following discussion is presented with the idea that, if one is going to include dynamic linking in a system and has a choice of processors, one should look for certain hardware features which are desirable in a dynamic linking environment. This section should not be viewed as a list of hardware support necessary for the feasible implementation of a dynamic linker.

A. HARDWARE FEATURES AFFECTING LINKER DESIGN

All the hardware features discussed in this section dictate in some manner how certain functions of a dynamic linker must be implemented. However, the first two features discussed (viz., indirect addressing and a hardware fault on indirection) are necessary to allow a linker to fully meet the design criteria of Table 1.

1. Hardware Indirection and Faults on Indirection

For the most part, it has been assumed the linker was invoked (on the first reference of an object) via the initialized code of the outgoing link. However, the most desirable method of linker invocation requires the processor to provide two hardware features: (1) The ability to reference data and call procedures using indirect addressing through memory; and (2) the ability to generate a hardware fault during indirection.

When a hardware fault on indirection is available, references to external objects are achieved via indirect addressing instructions where the final 'target address' (in the indirection sequence) is stored in the outgoing link of the executing procedure's linkage table. The outgoing link is initialized to cause a fault (on indirection) which results in the invocation of the linker as the fault handler. The linker snaps the outgoing link by altering the initialized fault-inducing code to either the virtual address of the incoming link (for external procedure calls) or the virtual address of the external data. (This represents the method used in Multics [11].)

Without a fault on indirection, it is not apparent how to pass external data as a parameter without first snapping the link to the data. This represents a violation of the delayed binding criterion (of Table 1) because the

binding of a symbolic name to a virtual address has been performed prior to first reference. (Note that even though the external data is passed as a parameter, it may not necessarily be referenced within the procedure.[15])

2. Other Features Influencing Linker Implementation

There are certain hardware features which do not restrict the implementation of a dynamic linker, but do effect certain aspects of the linker design. Two hardware features which are considered advantageous in a dynamic linking environment will be discussed.

The first feature relates to the number of segments available in a process address space. More specifically, if there are adequate segments (and each segment is of reasonable size), then it may not be necessary to frequently execute the unlinking portion of a dynamic linker. (Note that unlinking is still necessary because segments deleted from an address space should be unlinked.) This is considered advantageous since unlinking is considered one of the more expensive functions to execute. Note that if unlinking is not implemented, segments can always be conserved by combining smaller objects into a single segment and referencing each object via an entry point.

---

[15] One is free to judge how much of a limitation the absence of these two hardware features presents. However, the author does not consider it very prohibitive.

The object code produced by the translator is subject to the hardware features available. In a dynamic linking environment, some hardware features tend to simplify the object code produced for an external reference. For example, if hardware registers are automatically saved by the procedure CALL and RETURN conventions, then it is not necessary for the object code (during an external procedure call) to explicitly save and reset the linkage pointer. possess an indirect addressing CALL instruction but can only perform

B. HARDWARE FEATURES AFFECTING SYSTEM PERFORMANCE

There exist hardware capabilities which enhance system performance in a dynamic linking environment. These capabilities do not directly effect the design of the linker; but, because of the requirements dynamic linking places on the operating system (such as dynamic relocatability of code), the inclusion of certain hardware features serves to improve overall system performance.

In a dynamic linking environment, subroutines are not bound to virtual addresses (in a process address space) until run time. Therefore, they must reside on secondary storage in a relocatable form and be dynamically relocated during process execution. Thus, the more efficiently code can be relocated, the better system performance (viz., execution speed) will be. This implies that hardware

relocatability of code is desirable.

A second hardware capability which enhances system performance is hardware segmentation. Even though the linker design is not dependent on the support of segmentation hardware, many of the attributes associated with procedure and data objects (which are logical entities, or segments), are in fact intrinsic to segmentation. These attributes include object (unique) identifiers (viz., segment numbers) and object virtual addresses (viz., an object segment number + offset). It is therefore reasonable to conclude that segmentation hardware is desirable (but not essential) in a dynamic linking environment.

VII. A DEMONSTRATION OF DYNAMIC LINKING

In order to support the design concepts of this thesis, and, in a sense, prove the feasibility of microcomputer dynamic linking, a subset of the dynamic linker design (not including unlinking) was implemented on an Intel 8080 based system. The 8080 microprocessor [18] was selected because of its lack of hardware support, a fact which supported the contention that the linker design is hardware independent.

The implementation consisted of five modules: (1) a process initialization module, (2) the dynamic linker module, (3) the address space manager, (4) a display linkage table routine, and (5) a package of system library routines. Three of these modules (process initialization, the dynamic linker, and the address space manager) will be discussed in detail. The display linkage table routine was included in the implementation strictly to add clarity to the demonstration and will not be discussed in detail. (Source listings for the display linkage table routine and the system library routines are provided in appendix (B) for the interested reader.)

The implementation of the dynamic linker ran on the CP/M operating system [21]. The hardware support included two eight inch floppy disk drives and 65K of main memory. Modules were written in PL/M-80 [20] and compiled under the

Isis-II operating system [19].

(It should be noted at this time that because no translator which supported dynamic linking was available, test programs were hand compiled to produce the necessary object code, symbolic name tables, and linkage table templates.)

A. THE MODULES OF THE DYNAMIC LINKER

The three major modules of the linker were the process initialization module, the (dynamic) linker module, and the address space manager. Briefly, these modules perform the following functions:

Process Initialization is passed the argument 'program name' and performs the following:

1. Extracts the name of the program to be executed from the command line.

2. Causes the linker module and the address space manager to be initialized.

3. Causes the address space manager to (1) enter the program in the process address space and (2) load the program into memory.

4. Causes the linker module to build a linkage table for the program.

5. Builds the interrupt handler. The interrupt handler is invoked by initialized outgoing links and, in turn, invokes the linker module.

6. Starts the program in execution.

7. If the display toggle was set (in the command line), causes the process reference table and combined linkage table to be displayed following completion of program execution.

The linker module is invoked (by the fault handler) with the arguments ´linkage pointer´ and ´symbolic name offset´ (in the symbolic name table) and performs the following:

1. Extracts the character string name associated with the external reference from the calling procedure´s symbolic name table.

2. Invokes the address space manager passing as an argument the symbolic name of the external object (to be linked).

3. Builds a linkage table for the external object (if necessary).

4. Extracts the data associated with the entry name field (of the external reference) from the external object´s symbolic name table.

5. Snaps the outgoing and (if required) incoming links.

6. Causes the snapped outgoing link to be executed by returning the address of the outgoing link to the interrupt handler. The interrupt handler then jumps to the outgoing link.

The Address Space Manager consists of two submodules. ASM$Make$Accessable is invoked with the argument ´symbolic name´ (of an object) and performs the following:

1. Determines if the object is already in the process address space.

2. If not, loads the object into memory (performing a relocation if the object is executable code) and makes an entry for the object in the process reference table.

3. Returns to the point of call the unique identifier and base address (viz., 8080 'virtual address') of the object.

ASM$Remove$Seg is invoked with the argument 'symbolic name' and performs the following: 1. Removes an object from a process address space by deleting the object's entry in the process reference table.

The implementation of each of these modules will now be reviewed in detail. The discussion will include implementation details dictated by the 8080 hardware and CP/M operating system support utilized.

1. <u>Process Initialization</u>

The linker implementation was call 'Exec' and was invoked by the CP/M command line

```
A>Exec program_name $<cr>
```

The first function of process initialization was to scan the command line to determine the name of the program (viz., program_name) to be executed. This was performed by the READ$COMMAND$LINE subroutine which read the CP/M buffer to extract the program name. Additionally, if the last character of the command line was '$' (which is optional), the display toggle was set telling process initialization to display the process reference table and combined linkage

table following the completion of program execution. Additionally, since a program is executable code, READ$COMMAND$LINE assumes for the program a CP/M filetype of 'COM' [16].

Process initialization then calls on the subroutines INITIALIZE$ASM and INITIALIZE$LINKER which initialize the address space manager and linker modules respectively. (These two subroutines are a part of their respective modules and will be discussed in detail with the parent module.)

Having initialized the address space manager and linker module, process initialization then enters the program in the process address space and builds it a linkage table. The program is entered in the address space by calling on ASM$MAKE$ACCESSABLE (passing program_name as an argument.) ASM$MAKE$ACCESSABLE returns to process initialization the unique identifier and base address assigned to the program. (It should be noted that because the 8080 does not provide hardware segmentation, it was necessary to utilize a unique identifier and base address in

---

[16] CP/M utilizes a filetype field to distinguish the various types of files (on disk storage). The filetypes utilized by the linker implementation were (1) COM - a file of executable code; (2) DTA - an data file; (3) TMP - a linkage table template file; and (4) RIB - a file of relocation bits for a COM file.

place of the object segment number.) Process initialization then calls on an entry point into the linker module (viz.. the subroutine LINKAGE$TABLE$ROUTINES) which builds a linkage table for the program.

The next function of process initialization is to build the interrupt vector. It was decided to invoke the linker (when snapping a link) via a software fault. This technique allowed initialized outgoing links to be independent of the linker address by having the outgoing link jump (via a software fault) to a predetermined location which then invoked the linker. (The software fault used was an 8080 RST 4 instruction which saves the current execution point on the stack and jumps to the interrupt vector at location 20H.)

The interrupt vector first removes the return address placed on the stack by the RST 4 instruction. This address represents the address at the end of the outgoing link; when the link is snapped, it is desired to jump to the beginning of the outgoing link (to reference the external object). The next instruction of the interrupt vector calls the linker module passing to it the linkage pointer (the 8080 B and C register pair) and the offset (in the symbolic name table) of the entry for the object to be linked. (The symbolic name table offset is loaded in the D

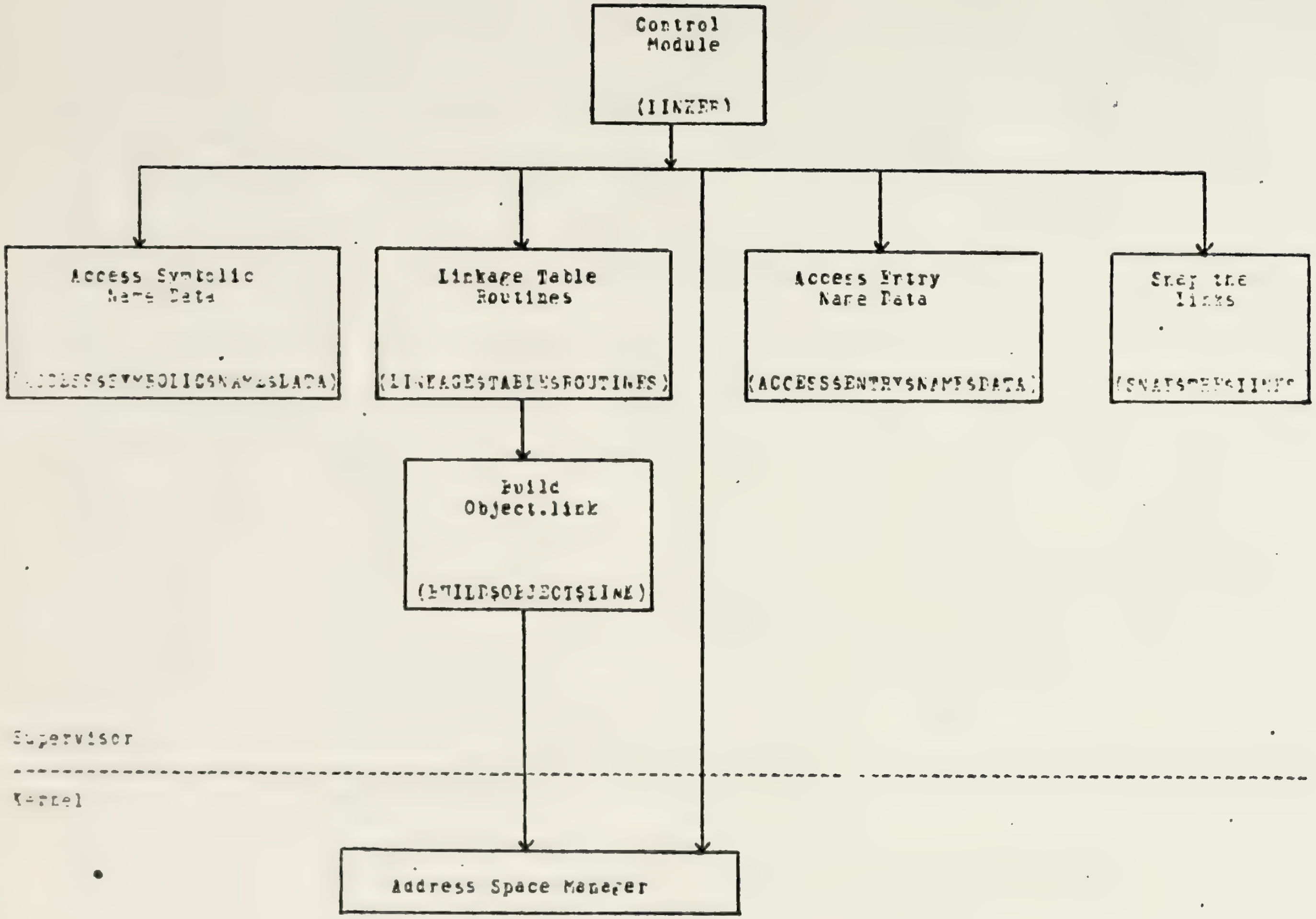
and E register pair by the initialized outgoing link.) When the linker module has completed execution it returns the address of the outgoing link to the interrupt vector (in the hardware H and L register pair). The interrupt vector then pops any arguments initially passed (by the caller) to the external object into the D and E register pair and jumps to the outgoing link. (The D and E register pair is used to pass arguments or pointers to a list of arguments between external objects.)

Finally, process initialization loads the initial value of the linkage pointer into the B and C register pair and invokes the program to be executed. These two functions are performed by the subroutine EXECUTE.

2. The Linker Module

The linker module was initially written in a high level pseudocode (appendix (A)) and then translated into PL/M-80. This permitted an orderly approach to the implementation of the dynamic linker module. The linker module consisted of five major subroutines and a control routine. The logical relation between linker subroutines is given in figure 13.

As has been noted the linker module (i.e., the control routine LINKER) was invoked by the interrupt vector. LINKER first calls on ACCESS$SYMBOLIC$NAME$DATA passing as



THE SUBROUTINES OF THE LINKER MODULE

FIGURE 13

arguments the linkage pointer and the symbolic name offset. ACCESS$SYMBOLIC$NAME$DATA utilizes the linkage pointer to address the linkage table of the calling procedure (which will be referred to as <Caller>) and extracts the address of Caller.sym. The entry of the external reference (viz., <Target|Entry_#1>) is then computed by adding the symbolic name offset to the address of Caller.sym.

ACCESS$SYMBOLIC$NAME$DATA can now extract (from the symbolic name table) the symbolic name "Target", the entry name "Entry_#1", the offset of <Target|Entry_#1>'s outgoing link (in Caller.link), and <Target>'s type (i.e., procedure or data). ACCESS$SYMBOLIC$NAME$DATA will compute the address of <Target|Entry_#1>'s outgoing link (by adding the outgoing link offset to the base of Caller.link). Next it will set the CP/M filetype for <Target> (viz., 'COM' for procedures and 'DTA' for data) in the symbolic name buffer. (The symbolic name buffer stores the filename and filetype of the object being linked in a standardized format. The standardized format is of the form 'FILENAME.FILETYPE'. Thus if <Target> was a procedure, the symbolic name buffer would contain the entry 'TARGET.COM'.)

LINKER can now call on the address space manager (ASM$MAKE$ACCESSABLE) to learn the segment number (i.e., the unique identifier and base address) of <Target>. Once LINKER

knows <Target>'s segment number data, it will invoke the subroutine LINKAGE$TABLE$ROUTINES.

LINKAGE$TABLE$ROUTINES determines if a linkage table already exist for <Target> by checking the valid entry bit of the linkage address table entry for <Target>. (Recall that the unique identifier of an object is used as a subscript into the linkage address table to access the base address of the object's linkage table.) If Target.link does not exist, LINKAGE$TABLE$ROUTINES will invoke BUILD$OBJECT$LINK to construct a linkage table for <Target> and will update <Target>'s entry in the linkage address table. Otherwise, LINKAGE$TABLE$ROUTINES merely returns a pointer (the parameter NEW$LINK$PTR) to point to Target.link.

BUILD$OBJECT$LINK first causes the address space manager to enter <Target>'s linkage table template in the process address space. It does this by appending to the program name (<Target>) the CP/M filetype of 'TMP'. (For example, if <Target> were a procedure, the executable code would exist in the file TARGET.COM while <Target>'s template is in the file TARGET.TMP.) Once the template is loaded into memory, BUILD$OBJECT$LINK first computes the address of Target.sym.

Recall that for a procedure, the symbolic name table is appended to the end of the object code. Thus, the address of the symbolic name table for procedures is computed by adding the offset of the symbolic name table (found in the template) to the base address of the object. For data, the symbolic name table is a part of the linkage table and its address is computed by adding the symbolic name table offset to the data object's linkage table base address.

BUILD$OBJECT$LINK then enters the (computed) symbolic name table address, the linkage table size, and the body of the linkage table in the combined linkage table as Target.link. (The combined linkage table was a statically allocated 1K block of memory.) BUILD$OBJECT$LINK then removes the template from the process address space by invoking ASM$REMOVE$SEG (an address space manager routine [17]).

Now that Target.link exist, the linker module can find Target.sym (via a pointer in Target.link's header) and

---

[17] The decision to build linkage tables in this manner was driven by an effort to simulate the mechanisms which would occur if hardware segmentation were available. To create Target.link in a segmented system, it would be necessary to make a copy of the (pure and sharable) template. However, in this implementation, since the disk copy of a template remains pure, the process copy (as introduced by the address space manager) could have just as easily served as the linkage table without recopying it into the combined linkage table. (Note that this approach would eliminate the need for a statically allocated combined linkage table.)

access the data associated with entry name. The routine ACCESS$ENTRY$NAME$DATA does this by searching Target.sym with the argument 'Entry_#1'. Recall that the symbolic name table entry for an entry name includes the incoming link offset and the entry point (of Entry_#1 into <Target>). Thus by adding the incoming link offset to the base address of Target.link, the incoming link address can be computed. Additionally, the entry point (offset) plus the base address of <Target> is the target address referenced by the symbolic name "Target|Entry_#1".

All the information necessary to snap the link is now available and LINKER calls on the subroutine SNAP$THE$LINKS to perform this function. The final subroutine of the linker module is INITIALIZE$LINKER which is invokes by process initialization. INITIALIZE$LINKER initializes various pointers (used by the linker module) and the valid entry bits of the linkage address table. It returns to process initialization the address of LINKER (for use in the interrupt vector), the address of the linkage address table (which is passed as a parameter to the display linkage table routine), and the base address of the combined linkage table (which is used in EXECUTE to initialize the linkage pointer).

3. The Address Space Manager Module

Because the CP/M operating system lacked any memory management executive, it was necessary for the address space manager to perform functions which would usually be provided by the operating system. Thus the address space manager had to be able to load objects into free memory and relocate executable code. These functions were carried out by the subroutines LOAD$OBJECT and RELOCATE respectively. The implementation of the two subroutines was extremely primitive providing only the minimum support necessary to allow the implementation of the remainder of the address space manager (and will not be discussed in any further detail).

Like the dynamic linker module, the address space manager was first written in pseudocode (appendix (A)) and translated into PL/M-80. It centers around the managing of the process reference table which is implemented as an array of structures of the form:

```
Process_Reference_Table : ARRAY of STRUCTURES of
                            Valid_bit : BOOLEAN;
                            Name : ARRAY of CHARACTERS;
                            Base_address : ADDRESS;
                          END;
```

The valid_bit field was set to 'valid' if the entry

represented an object in the process address space. The name field contained the object name in standardized form (e.g., CALLER.COM) while the base_address is the location (in memory) where the object was loaded. Note also that an object's unique identifier represents an implied process reference table field and corresponded to the subscript of the object's entry in the process reference table.

When ASM$MAKE$ACCESSABLE is invoked, it is passed the object name (in standard form) as an argument. ASM$MAKE$ACCESSABLE first searches the process reference table to determine if <Target> already has an entry (implying <Target> is already in the process address space). If not, LOAD$OBJECT is invoked to load <Target> into memory returning the base address of <Target> to the point of call. ASM$MAKE$ACCESSABLE then enters <Target> in the process reference table in the first free entry. The final function of ASM$MAKE$ACCESSABLE is to return the base address and unique identifier (viz., the process reference table subscript) of <Target> to the point of call.

The subroutine ASM$REMOVE$SEG is passed an object name (in standard form) and deletes the object from the process address space by setting the object's valid_bit in the process reference table to 'invalid'.

Two other subroutines included in the address space manager were DISPLAY$PRT which displayed the process reference table (and is not necessary in a dynamic linker implementation) and INITIALIZE$ASM. INITIALIZE$ASM is invoked by process initialization (as is DISPLAY$PRT) and initializes the valid_bits of the process reference table to ´invalid´. Additionally it statically sets the size of the process reference table (which was arbitrarily set to 16 entries) and initializes a free memory pointer for the LOAD$OBJECT subroutine.

B. THE TEST PROGRAMS

Two test programs were run on the dynamic linker. The first, DEMO, computed and displayed (in hexadecimal form) the multiplication and addition tables (with appropriate headers for the numbers from 0 to 15. The second test program, SUM, added the elements of an external data array and displayed the result in hexadecimal form. DEMO demonstrated all the capabilities desired of dynamically linked objects. SUM was included to provide a simple example that will be explained in detail.

1. Test Program Construction

Before discussing either test program further, it is useful to explain the mechanics used in their construction. First, because a translator which supported dynamic linking was not available, it was necessary to hand assemble those portions of the test programs unique to dynamic linking. These included translated external references, symbolic name tables, and linkage table templates[18]. All test program source listings and program test results are included in appendix (C).

---

[18] The test programs, including templates, symbolic name tables, and relocation bits (for executable code) were written in 8080 assembly code and assembled using the Digital Research 8080 Assembler [17].

a. The Assembled Symbolic Name Table

The symbolic name table of an object can be found (in the source listings) at the end of either the object code (for procedures) or in the linkage table template (for data). Each entry in the symbolic name table consist of four field. For clarity, each field was preceded by a label. Entries were of the following form:

```
DESCn  : DB byte_1 [19]
LINKn  : DB low_byte, high_byte
ENTRYn : DB low_byte, high_byte
NAMEn  : DB 'OBJECT_NAME:ENTRY_NAME' or 'ENTRY_NAME'
```

DESCn represents the entry descriptor (of the nth symbolic name table entry). The most significant bit of byte_1 indicated the object type (viz., 0 for procedures and 1 for data). The five least significant bits of byte_1 contained the number of characters in the name field. The remaining two bits of DESCn were unused.

LINKn is the offset of the entry's outgoing or incoming link in the parent object's linkage table.[20] The ENTRYn field is an entry point offset in the parent object

---

[19] DB is an assembler pseudo-operator that tells the assembler that the rest of the line represents data. Data not surrounded by single quotes is translated as a numerical value while data in quotes is an ASCII character string.

[20] In the 8080, two byte values are stored in memory with the low byte in the lower numbered memory location. Thus the number 1020H would appear as 20H, 10H when used in a DB field.

associated with some entry name. For an external reference, low_byte and high_byte of this field were arbitrarily set to zero.

The NAMEn field held the symbolic name associated with the entry. This field contained either an entry name (e.g., ENTRY_#1), or the name of an external reference (e.g., OBJECT_NAME:ENTRY_NAME). For the NAME field of an external reference, the 'ENTRY_NAME' portion is optional. When left out, it implies that the entry name to be used is the same as the object name. For example, the procedure MULT has an entry point by the same name but appears as 'MULT' in DEMO's symbolic name table (vice 'MULT:MULT').

b. The Assembled Template

The linkage table template was constructed as assembled code. Templates were of the form:

```
SIZE : DB low_byte, high_byte
SNT  : DB low_byte, high_byte

BODY : DB 00, 00, 00, 00, 00, 00        (incoming link)

       PUSH D                           (outgoing link)
       LXI D, symbolic_name_table_offset
       RST 4
```

The SIZE field contains the number of bytes in the template. SNT represents the offset (i.e., number of bytes) of the symbolic name table from the beginning of either a procedure segment or a data segment's template.

The BODY of a template contains two types of entries. For an incoming link, the template merely reserves six bytes (initialized to 0) in the combined linkage table in which the snapped incoming link will eventually be placed. An outgoing link consist of three assembly code instructions. The first instruction (PUSH D) saves the argument register (viz., the D and E register pair) prior to loading that register with the symbolic name table offset of the external object to be linked. The third outgoing link instruction (RST 4) causes a software fault resulting in the invocation of the linker via the interrupt vector.

c. Other Problems in Test Program Construction

Because the 8080 microprocessor does not have an indirect addressing CALL instruction, the transfer of control to an outgoing link (by the executing procedure) deserves explanation. Recall that it is desired to perform the following:

```
    CALL (Lp + outgoing_link_offset)
```

To achieve this in 8080 code, the following sequence of instructions was used:

```
                   PUSH B
                   LXI H, return_address
                   PUSH H
                   LXI H, outgoing_link_offset
                   DAD B
                   PCHL
 return_address :  POP B
```

The first instruction (PUSH B) saves the linkage pointer. The next two instructions save the return address (which is normally done automatically by a CALL instruction). The H and L register pair is then loaded with the outgoing_link_offset and added to the B and C register pair (viz., the linkage pointer) by the DAD B instruction. DAD B adds the B and C registers to the H and L registers and leaves the result in the H and L registers. The value 'Lp + outgoing_link_offset' is in now jumped to by the PCHL instruction (which transfers control to the address stored in the H and L registers). The final instruction (POP B) restores the linkage pointer upon return from the external procedure.

Very briefly, a relocation bits file was constructed by hand and was of the following form:

```
SIZE  : DB low_byte, high_byte
L0100 : DB binary_number_1, binary_number_2
```

The SIZE field represents the number of bytes in the relocation bits file. The remainder (of the file) consisted of two binary numbers preceded by a label such as L0100 (where 0100 corresponds to an address in the procedure object code listing). A '0' in a binary number corresponds to a non-relocatable byte of object code. A '1' identifies the byte as the first of a two byte relocatable address.

2. The Test Program DEMO

The address space of DEMO included four objects: (1) the procedure segment DEMO(nstration); (2) the procedure segment MULT(iply) which included the entry point 'MULT'; (3) the procedure segment DISPLY which included the entry points 'HEX_VALUE' and 'BUFFER'; (4) and the data segment HEADER which included the entry points 'HEADER' and 'TITLE'.

As has been noted, DEMO computed and displayed the (hexidecimal) multiplication and addition tables for the values 0 through 15. The construction of each table was performed by the internal (to DEMO) procedure Build_table which is passed a subroutine as a parameter (viz., ADD, an internal procedure, and MULT, an external procedure). ADD and MULT are passed (by Build_table) a number that is added/multiplied by 0 through 15. The result of the computation is displayed by invoking the external procedure DISPLY.HEX_VALUE. Thus to build a hexadecimal table Build_table simply invokes either ADD or MULT sixteen times passing as a parameter the values from 0 to 15.

Before building a table, DEMO displays an appropriate heading. It does this by dynamically linking to the data segment HEADER, inserting the appropriate title (viz., MULTIPLICATION or ADDITION) at the entry point HEADER.TITLE, and then displaying HEADER by passing it as an

argument to the external procedure DISPLY.BUFFER.

The dynamic linking which takes place during the execution of DEMO is given in figure 14. DEMO includes examples of all the various capabilities (of external objects) desired in a dynamic linking environment including:

(1) The ability to dynamically link and execute external procedures--DEMO dynamically links to and invokes DISPLY.

(2) The ability to reference external data--DEMO links to and references HEADER.

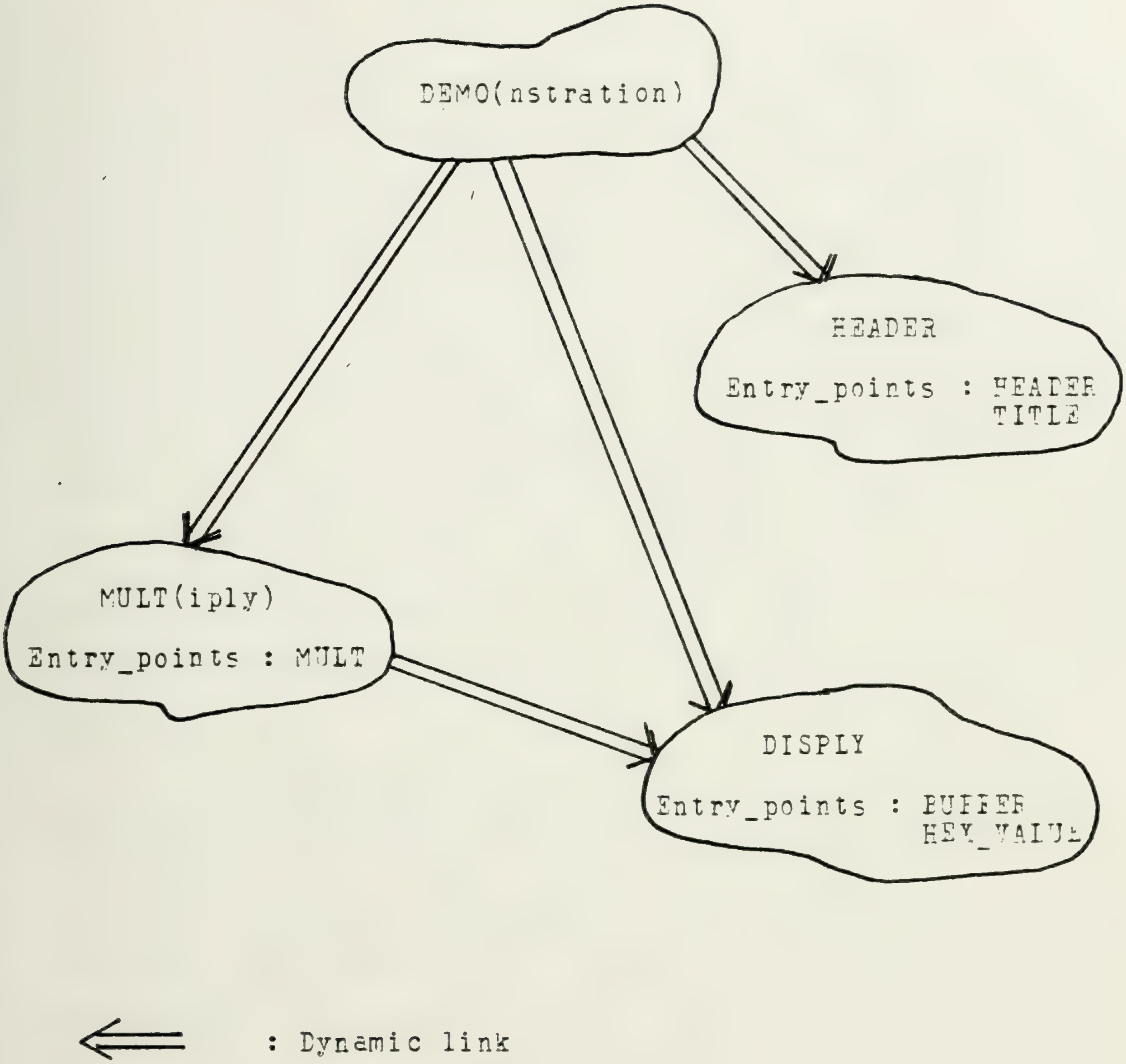
(3) The ability to pass external objects as arguments--HEADER and MULT are passed to DISPLY and Build_table respectively.

(4) The ability of an external object to engage in dynamic linking--MULT dynamically links to DISPLY.HEX_VALUE.

(5) The implementation of entry points in objects--DISPLY and HEADER both are referenced via entry points.

3. The Test Program SUM

The procedure SUM was included to allow a complete and comprehensive discussion of the concepts presented in this thesis. SUM itself is rather simple. It dynamically links to the external data segment ARRAY and sums the (data) bytes of ARRAY. The results are displayed by dynamically linking to DISPLY.HEX_VALUE (passing the sum of ARRAY's bytes as an argument). DISPLY.BUFFER is also invoked to display appropriate messages along with the computation result.



DYNAMIC LINKING IN DEMO

FIGURE 14

A pseudocode listing of SUM is given in figure 15 while figure 16 presents a representative assembly code translation of SUM. The assembly code used is not associated with any particular microprocessor, but is considered within the capabilities of most microprocessor instruction sets. The only instruction used which may cause confusion is LDPARAM (viz., load parameter register). This instruction is simply a register load but the pneumonic LDPARAM is offered to signify the passing of arguments to an external procedure. (Note that the dynamic linker demonstration implementation uses the D and E register pair for this purpose.)

The combined linkage table for SUM is shown in figure 17. (The figure does not include ARRAY.link or a link for DISPLY.BUFFER). The linkage table for SUM includes an incoming link (entry #1) which would be used if SUM were referenced as an external object. Entry #2 is the outgoing link from SUM to ARRAY while Entry #3 represents the outgoing link from SUM to DISPLY.HEX_VALUE.

When the two outgoing links of SUM are snapped, the unlinking data is included in the snapped link and includes the symbolic name table offset of ARRAY and DISPLY.HEX_VALUE (in SUM.sym) respectively and the appropriate linked list pointers. Unlinking linked lists are implemented as circular linked list. Thus the linked list for DISPLY starting with

```
PROCEDURE Sum;

   /* Sum adds the bytes of the external data structure
      'Array' and then calls on the external procedure
      'Disply' to output the result. */

   DECLARE Sum ENTRY POINT;
           Array DATA EXTERNAL;
           Disply PROCEDURE EXTERNAL;

           result : BYTE;
           array_pointer : POINTER:
           data_array BASED at array_pointer STRUCTURE of
             number_of_bytes : BYTE;
             data : ARRAY of BYTES;
             END;

           i : BYTE;

   /* end of declarations */

   array_pointer = address of array;
   result = 0;

   FOR i = 1 to data_array.number_of_bytes;
     result = result + data_array.data (i);
   ENDFOR;

   CALL disply.buffer ('The sum of the data array is ','&');
   CALL disply.hex_buffer (result);

   /* generate a carriage return and line feed */

   CALL disply.buffer (CR, LF, '&');
   CALL disply.buffer ('End of Sum','&');

END Sum,
```

PSEUDOCODE FOR SUM

FIGURE 15

OBJECT CODE

```
    /* code */                                              /* comments */

  CALL (lp - outgoing_link_offset_1)                      /* dynamically link to array
  LOAD array_pointer, ext. data pointer                   /* load array_pointer with the value in the
                                                             external data pointer (address of array
  LOAD (result),0                                         /* initialize result to 0
  LOAD (i),1                                              /* initialize i to 1
label loop:
  COMPARE i to data_array.number_of_bytes
  JUMP to endloop if i > data_array.number_of_bytes
  LOAD accumulator,result                                 /* load the accumulator with result
  ADD data_array.data (i) to accum
  STORE accumulator in result                             /* result = result + data_array.data (i)
  i = i + 1
  JUMP to loop
label endloop:
  LDPARAM string_1_address                                /* load the parameter register with string 1's address
  PUSH lp                                                 /* save the linkage pointer
  CALL (lp + outgoing_link_offset_2)                      /* dynamically link and invoke disply.buffer
  POP lp                                                  /* restore the linkage pointer
  LDPARAM result                                          /* load the parameter register with result
  PUSH lp                                                 /* save the linkage pointer
  CALL (lp + outgoing_link_offset_3)                      /* dynamically link and invoke disply.hex_value
  POP lp                                                  /* restore the linkage pointer
  LDPARAM string_address_2                                /* load the parameter register with string 2's address
  PUSH lp                                                 /* save the linkage pointer
  CALL (lp + outgoing_link_offset_2)                      /* invoke disply.buffer
  POP lp                                                  /* restore the linkage pointer
  LDPARAM string_address_3                                /* load the parameter register with string 3's address
  PUSH lp                                                 /* save the linkage pointer
  CALL (lp + outgoing_link_offset_2)                      /* invoke disply.buffer
  POP lp                                                  /* restore the linkage pointer
  RET                                                     /* end of sum

/* data declarations */

string_1 = 'The sum of the data array is ', delimiter
string_2 = ASCII carriage return, ASCII line feed, delimiter
string_3 = 'End of sum', delimiter

/* symbolic name table */

   [descriptor_1], [ incoming_link_offset ], [entry_point_1], sum
   [descriptor_2], [outgoing_link_offset_1], [     void     ], array
   [descriptor_3], [outgoing_link_offset_2], [     void     ], disply.buffer
   [descriptor_4], [outgoing_link_offset_3], [     void     ], disply.hex_value
```

OBJECT CODE FOR SUM

FIGURE 16

LINKAGE TABLE

```
offset     before execution                 after execution
------     ----------------                 ---------------

                      /* Linkage Address Table */
          ------------------          ------------------
 00       |     sum lp       |        |     sum lp       |   /* offset 04
 01       |      nil         |        |    array lp      |   /* not shown
 02       |      nil         |        |   disply lp      |   /* offset 0E
 03       |      nil         |        |      nil         |
          ------------------          ------------------

                      /* Sum Linkage Table */
          ------------------          ------------------
 04       | linkage tbl size |        | linkage tbl size |
          |..................|        |..................|
 05       | sym name tbl addr|  header | sym name tbl addr|
          |..................|        |..................|
 06       |  linked list ptr |        |  linked list ptr |   /* offset 00
          ------------------          ------------------
 07       |unsnapped incoming|        |unsnapped incoming|
          |..................| entry #1 |..................|
 08       |   link to Sum    |        |   link to Sum    |
          ------------------          ------------------
 09       |LIBPARAM SNT offset|       |LOAD ptr reg,addr |   /* snapped link to array
          |..................|        |..................|
 0A       |  INVOKE LINKER   | entry #2 |      RET         |
          |..................|        |..................|
 0B       |     filler       |        | # |  list ptr    |   /* unlinking data (SNT offset, ptr)
          ------------------          ------------------
 0C       |LIBPARAM SNT offset|       |JUMP incoming link|   /* of disply.buffer,  offset 11
          |..................| entry #3 |..................|
 0D       |  INVOKE LINKER   |        | # | list ptr (02)|   /* unlinking data (SNT offset, ptr)
          ------------------          ------------------

                                      /* Disply Linkage Table */
                                      ------------------
 0E                                   | linkage tbl size |
                                      |..................|
 0F                           header  | sym name tbl addr|   /* of disply
                                      |..................|
 10                                   |  linked list ptr |   /* offset 0C
                                      ------------------
 11                                   |1| LOAD lp, offset|   /* load lp with offset 0E
                            entry # 1 |..................|
 12                                   |JUMP Disply.hex_val|  /* snapped incoming link
                                      ------------------
```

COMBINED LINKAGE TABLE FOR SUM

FIGURE 17

the header entry (in DISPLY.link) goes from offset 10H to 0CH (the snapped outgoing link from SUM to DISPLY.HEX_VALUE). The linked list pointer at 0DH (in SUM.link) points to DISPLY's linkage address table entry which in turn points to DISPLY.link (viz., DISPLY.link's header which contains the first node of DISPLY's linked list).

The assembly code for SUM, ARRAY, and DISPLY is included in appendix (C) along with the output generated by SUM, the process reference table, and the combined linkage table also. The process reference table and combined linkage table are annotated to provide additional clarification.

4. Observations on the Implementation

a. Size of the Dynamic Linker Implementation

The dynamic linker including the display linkage table and display process reference table routines was 8320 bytes in length. This includes 1K bytes of memory statically allocated to the combined linkage table and 150 bytes reserved for the hardware stack. (It should be noted that additional memory was allocated to the PL/M-80 stack segment to prevent stack overflow during test program execution. This was necessary since the PL/M-80 stack is allocated based on the needs of the dynamic linker and does not take into account stack operations done by other procedures in a process.) It is emphasized that no effort was make to

optimize the object code. Instead, the dynamic linker was written to be as clear and obvious as possible.

The dynamic linker was also compiled without the display linkage table and display process reference table routines (which were included for the purposes of the demonstration only). This edition of the linker was 6272 bytes in length. It is estimated that a complete (i.e., including unlinking) and optimized implementation of the dynamic linker should require about 7000 bytes of object code. It is noted that error conditions were not checked for by the dynamic linker. However, since there are essentially only two error conditions which could occur, it is felt that the size estimate for a dynamic linker is still valid. The error conditions which may occur are (1) a reference is made to a non-existant entry point (References to non-existant files are flagged by the library routines.), and (2) The statically allocated 1K combined linkage table is overflowed. Such problems as running out of free memory or process reference table entries are handled by the unlinker.

b. Overhead Associated with Snapped Links

One of the major arguments against dynamic linking is the issue of overhead associated with snapped links. Before debating this must besue, it is observed that the cost of dynamic linking associated with snapping a link (i.e., the first reference of an external object) is on the

order of the overhead required to statically link the same object.

With respect to snapped procedure links, the overhead (associated with the linker implementation) lies in two areas. First, the linkage pointer must be updated to always indicate the executing procedure's linkage table. Thus the linkage pointer must be saved and restored for each external procedure reference, which requires an additional two instructions. Additionally the linkage pointer is set to point to the (dynamically linked) external procedure by the snapped incoming link, which requires a third instruction. Secondly, the execution point goes from the calling procedure to the external procedure via the snapped outgoing and incoming links. This requires two jump instructions not needed for internal procedure calls thereby bringing the total overhead to five instructions. It is noted that the extensive code necessary in invoke an external object's outgoing link is considered a limitation of the 8080 (because of the lack of an 8080 indirect call instruction) and is not considered overhead induced by dynamic linking.

Recall that to reference external data (via the outgoing link) a call to the outgoing link is performed, the virtual address of the data is loaded in a pointer, and a return instruction (to the calling procedure) is executed. Since internal data is essentially referenced by loading a

pointer with the address of the (internal) data, the overhead associated with dynamic linking (for data) is limited to a CALL and RETURN instruction.

VIII. CONCLUSIONS

Based on the research supported in this thesis it is reasonable to assert that dynamic linking is feasible in a microcomputer environment. However, given that the linker design is implementable on microprocessors, it can be asserted that dynamic linking does not require the support of specialized hardware and thus can be feasibly implemented on most general purpose computers (including minicomputers and main frames). The overhead is within reason and can be far outweighed by the derived benefits. It has been implied [9, 13, 14] that dynamic linking requires the support of specialized hardware. It is felt that the major contribution of this thesis is to dispell that notion.

APPENDIX A - PSEUDOCODE

```
EXECUTIVE Linker;

 /*
     Explanation of variables and constants :

       En_buffer - The entry name buffer is a string variable
         where  the  entry  name  associated with an external
         reference is stored once the  entry  name  has  been
         extracted from the calling procedure's symbolic name
         table.

       Fixed_Sn_offset  - The fixed symbolic name offset is a
         constant which represents the  number  of  bytes  in
         that  portion  of  a  symbolic name table entry that
         does not vary in size (i.e.,  the  descriptor,  link
         offset, and entry point).

       Free_link_table  -  The free linkage table variable is
         the next free location in the combined linkage table
         where   new   (object)   linkage   tables   can   be
         constructed.

       In_link_address  -  The  incoming  link address is the
         virtual  address  of  the  incoming  link  for   the
         <external_procedure|entry_name> being linked.

       Incoming_link - The incoming link structure represents
         the  format  of  an  incoming link. Incoming link is
         based at the incoming link address.

       Linkage_ptr - The linkage pointer.

       Linkage_array  -  Linkage_array  is  a  linkage  table
         structure based at the linkage pointer.

       New_link_ptr - The new linkage pointer is assigned the
         value  of the linkage pointer of the external object
         being linked.

       New_link_table - The new linkage table  is  a  linkage
         table  structure  (of  the  external  object  begin
         linked) based at the new linkage pointer.

       Object_seg_number  -  Object  segment  number  is  the
         segment number assigned to the external object being
         linked.
```

Object_type - Object type represents whether the external object begin linked is a procedure or data.

Out_link_address - The outgoing link address is the virtual address of the outgoing link assigned to the <external_object|entry_name> being linked.

Outgoing_link - The outgoing link based at the outgoing link address.

Sn_buffer - The symbolic name buffer is a string variable where the symbolic name of the external reference is stored once the symbolic name has been extracted from the calling procedure's symbolic name table.

Sn_address - The symbolic name address is a pointer into a symbolic name table.

Sn_item - A symbolic name item is a structure based at the symbolic name address and represents an entry in a symbolic name table.

Sn_offset - The symbolic name offset is the parameter passed the linker and is the offset into the calling procedure's symbolic name table of the external reference to be linked.

Sn_size - The symbolic name size is the number of character in an <external_reference|entry_name> as found in a symbolic name table entry.

Sn_size_mask - The symbolic name size mask is used to extract the size of a symbolic name from a descriptor in the symbolic name table of the calling procedure.

Sn_type_mask - The symbolic type mask extracts the type of an external reference (i.e., procedure or data) from the descriptor.

Target_address - The target address is the ultimate virtual address in an external object which the calling procedure seeks to reference.

Template_seg_number - The template segment number is the segment number assigned to the linkage table template when it is entered in a process address space.

Template - Template is a linkage table template structure based at template segment number.

Type_data - Type data is a constant which is used to identify external data objects.

Type_procedure - Type identifies external procedure procedure objects.

/* end of variable explanations */

/* explanation of declaration types */

ADDRESS - a virtual address.
BYTE - the contents of a virtual address.
CHARACTER - an ASCII character.
INTEGER - a variable.
POINTER - an address variable which points to a user defined data structure.
STRUCTURE - a Pascal record.

/* end of explanation of declaration types */

```
/* The following is a list of variable and constant
   declarations used in the Linker.   */

DECLARE


   En_buffer : STRUCTURE of
               size : INTEGER;
               name : ARRAY of CHARACTERS;
               END;

   Fixed_Sn_offset : INTEGER CONSTANT;
   Free_link_table : ADDRESS;

   In_link_address : POINTER;
   Incoming_link : STRUCTURE BASED at In_link_address of
                   Link_snapped_bit : BYTE;
                   Load_Lp : INTEGER;
                   Jump_inst : INTEGER;
                   END;

   Linkage_ptr : POINTER;
   Linkage_array : STRUCTURE BASED at Linkage_ptr of
                   Size : INTEGER;
                   Snt_address : ADDRESS;
                   Body : ARRAY of BYTE;
                   END;

   Linkage_address_table : ARRAY of ADDRESS;

   New_link_ptr : POINTER;
   New_link_table : STRUCTURE BASED at New_link_ptr of
                    Size : INTEGER;
                    Snt_address : ADDRESS;
                    Body : ARRAY of BYTE;
                    END;

   Object_seg_number : ADDRESS;
   Object_type : BYTE;

   Out_link_address : POINTER;
   Outgoing_link : ARRAY of INTEGER
                      BASED at Out_link_address;

   Sn_buffer : STRUCTURE of
               size : INTEGER;
               name : ARRAY of CHARACTERS;
               END;
```

```
   Sn_address : POINTER;
   Sn_item : STRUCTURE BASED at Sn_address of
             descriptor : BYTE;
             name : ARRAY of CHARACTERS;
             link_offset : INTEGER;
             entry_point : INTEGER;
             END;

   Sn_offset : INTEGER;
   Sn_size : INTEGER;
   Sn_size_mask : BYTE CONSTANT;
   Sn_type_mask : BYTE CONSTANT;

   Target_address : ADDRESS;

   Template_seg_number : POINTER;
   Template : STRUCTURE BASED at Template_seg_number of
              Size : INTEGER;
              Snt_offset : INTEGER;
              Body : ARRAY of BYTE;
              END;

   Type_data : BYTE CONSTANT;
   Type_procedure : BYTE CONSTANT;

END of DECLARATIONS;
```

```
/* Linker Control Module */

BEGIN

  /* Save processor registers if necessary */

  CALL Access_Symbolic_Name_Data;
     PARAMETER_LIST : Sn_offset, Sn_buffer, En_buffer,
                      Linkage_pointer;
     RETURN_LIST    : Sn_address, En_buffer, Sn_buffer,
                      Object_type, Out_link_address;


  /* ASM_Make_Accessable calls on the Address Space Manager
     to add the object found in Sn_buffer.name to the
     process address space and return the segment
     number assigned to that object. */

  CALL ASM_Make_Accessable;
     PARAMETER_LIST : Sn_buffer.name;
     RETURN_LIST    : Object_seg_number;

  CALL Linkage_Table_Routines;
     PARAMETER_LIST : Object_seg_number, Link_address_table,
                      Free_link_table, New_link_ptr;
     RETURN_LIST    : New_link_ptr, Link_address_table,
                      Free_link_table;

  CALL Access_Entry_Name_Data;
     PARAMETER_LIST : Sn_address, En_buffer, New_link_ptr,
                      Object_type, Object_seg_number;
     RETURN_LIST    : Target_address, In_link_address;

  CALL Snap_the_Links;
     PARAMETER_LIST : In_link_address, Out_link_address,
                      New_link_ptr, Object_type,
                      Target_address;
     RETURN_LIST    : None;


  /* restore processor registers if necessary */

  JUMP to Out_link_address;

END Linker;
```

```
/***********************************************************

Access_Symbolic_Name_Data performs the following functions:

 1.  Obtains the address of the symbolic name of the
     external reference being linked.

 2.  Loads the symbolic name of the external reference
     in the symbolic name buffer (Sn_Buffer).

 3.  Loads the entry name of the external reference
     in the entry name buffer (En_buffer).

 4.  Computes the outgoing link address and determines
     whether the external object is a procedure or data.

***********************************************************/

PROCEDURE Access_Symbolic_Name_Data;

  PARAMETER_LIST : Sn_offset, Sn_buffer, En_buffer,
                   Linkage_pointer;

    DECLARE i, temp : INTEGER;

    Sn_address = Linkage_array.Snt_address + Sn_offset;
    Sn_size = Sn_item.size AND Sn_size_mask;

    /* Load the symbolic name into Sn_buffer.name. */

    i = 1;
    DO WHILE (Sn_item.name(i) <> ':') AND (i <= Sn_size);
        Sn_buffer.name(i) = Sn_item.name(i);
        i = i + 1;
    ENDWHILE;

    /* Load symbolic name size into Sn_buffer.size. */

    IF i = Sn_size THEN Sn_buffer.size = i;
      ELSE Sn_buffer.size = (i-1);
```

```
     /* Load the entry name buffer with the entry name. */

     IF i = Sn_size THEN BEGIN

        /* No entry name specified, default to the
           symbolic name as the entry name.  */

         En_buffer.size = Sn_size;
         FOR i = 1 to Sn_size by 1 DO
           En_buffer.name(i) = Sn_item.name(i);

       ENDTHEN;

     ELSE BEGIN   /* entry name specified */
         temp = i;
         i = i + 1;

         /* load size of entry name in En_buffer.size */

         En_Buffer.size = Sn_size - i;

         /* load entry name into entry name buffer */

         DO WHILE (i <= Sn_size)
             En_Buffer.name(i - temp) = Sn_item.name(i);
             i = i + 1;
         ENDWHILE;

    ENDELSE;

      /* Compute the address of the outgoing link and
         determine the type (procedure or data) of the
         external object. */

      Out_link_address = Linkage_pointer +
                         Sn_item.link_offset;
      Object_type = Sn_item.descriptor AND Sn_type_mask;


   RETURN_LIST : Sn_address, En_buffer, Sn_buffer,
                 Object_type, Out_link_address;

END Access_Symbolic_Name_Data;
```

```
/************************************************************

Linkage_Table_Routines performs the following functions:

 1.  Determines if a linkage table already exist for the
     external reference being linked.

      a.  If not, Linkage_Table_Routines initialized the
          Linkage Address Table value for the object and
          then calls on Build_Object.link.

      b.  If so, Linkage_Table_Routines sets a return
          parameter (New_link_ptr) equal to the linkage
          pointer value for the new object's linkage table.

 ************************************************************/

 PROCEDURE Linkage_Table_Routines;

   PARAMETER_LIST :  Object_seg_number, Link_address_table,
                     Free_link_table, New_link_ptr;

     IF Link_address_table (Object_seg_number) = nil THEN

       /* This is the first time the object has been
          referenced by the process and the linker must
          build a linkage table for the object.  */

     BEGIN
         Link_address_table (Object_seg_number) =
                                             Free_link_table;
         CALL Build_Object.link;
            PARAMETER_LIST : Object_type, Free_link_table,
                             Sn_buffer, Object_seg_number;
            RETURN_LIST    : New_link_ptr, Free_link_table;

     ENDTHEN;

     ELSE

        /* The object already has a linkage table. */

       New_link_ptr = Link_address_table (Object_seg_number);

   RETURN_LIST : New_link_ptr, Link_address_table,
                 Free_link_table;

 END Linkage_Table_Routines;
```

```
/****************************************************************

Build_Object.link performs the following functions:

 1.  Causes the Address Space Manager to load the external
     object's linkage table template into the process
     address space.

 2.  Initializes a return parameter (New_link_ptr) to the
     value of the object's linkage pointer.

 3.  Appends Object.link to the end of the combined linkage
     table.

 4.  Deletes the linkage table template from the process
     address space.

****************************************************************/

PROCEDURE Build_Object.link;

  PARAMETER_LIST : Object_type, Free_link_table, Sn_buffer,
                   Object_seg_number;

    DECLARE i : INTEGER;

    /*  The following two steps cause Sn_buffer.name to be
        loaded with the filename <symbolic name.template>
        and then invokes the Address Space Manager to have
        the template loaded into the process address space
        (with ASM_Make_Accessable returning the segment
        number assigned to the template.  */

    APPEND 'template' to Sn_buffer.name;
    CALL ASM_Make_Accessable;
       PARAMETER_LIST : Sn_buffer.name;
       RETURN_LIST    : Template_seg_number;

    New_link_ptr = Free_link_table;

    IF Object_type = Type_procedure THEN BEGIN

       /* If the object is a procedure, then its symbolic
          name table is in the object code segment. */

        New_link_table.Snt_address = Template.Snt_offset +
                                     Object_seg_number;
    ENDTHEN;
```

```
    ELSE BEGIN

       /* If the object is data, then its symbolic name
          table is in its template. */

        New_link_table.Snt_address = New_link_ptr +
                                       Template.Snt_offset;
    ENDELSE;

    New_link_table.Size = Template.Size;

    FOR i = 0 to (Template.Size - 2) by 1 DO
        New_link_table.Body (i) = Template.Body (i);
    ENDFOR;

    Free_link_table = Free_link_table + 1 +
                               Template.Size;

    CALL ASM_Remove_Seg;
       PARAMETER_LIST : Sn_buffer.name;
       RETURN_LIST    : None;

  RETURN_LIST : New_link_ptr, Free_link_table;

END Build_Object.link;
```

```
/*********************************************************************

Access_Entry_Name_Data performs the following functions:

 1.  Computes the target address in the external object to
     be utilized in the linkage process.

 2.  Computes the incoming link address (if applicable).

*********************************************************************/

PROCEDURE Access_Entry_Name_Data;

  PARAMETER_LIST : Sn_address, En_buffer, New_link_ptr,
                   Object_type, Object_seg_number;

    /* Get_Next_Sn_item causes Sn_address to point to the
       next entry in the external object's Symbolic Name
       Table.   */

    PROCEDURE Get_Next_Sn_item (Sn_address);
      Sn_address = Sn_address + Fixed_Sn_offset
                   + (Sn_item.descriptor AND Sn_size_mask);
      RETURN Sn_address;
    END Get_Next_Sn_item;

    /*  Begin Access_Entry_Name_Data.  */

    Sn_address = New_link_table.Snt_address;

    DO WHILE Sn_item.name <> En_buffer.name;
      CALL Get_Next_Sn_item;

    Target_address = Object_seg_number + Sn_item.entry_point;

    IF Object_type = Type_procedure THEN
      In_link_address = New_link_ptr + Sn_item.link_offset;

    RETURN_LIST : Target_address, In_link_address;

END Access_Entry_Name_Data;
```

```
/*************************************************************

Snap_the_Links performs the following functions:

 1.  Snaps the outgoing link and incoming link for a
     procedure object.

 2.  Snaps the outgoing link for a data object.

*************************************************************/

PROCEDURE Snap_the_Links;

  PARAMETER_LIST : In_link_address, Out_link_address,
                   New_link_ptr, Object_type,
                   Target_address;

    IF Object_type = Type_procedure THEN BEGIN

       /* Snap a link for an external procedure. */

        Outgoing_link (0) = 'Jump to' In_link_address;

        IF Incoming_link.link_snapped_bit is unsnapped THEN
        BEGIN
            Incoming_link.Load_Lp = 'Load Lp' New_link_ptr;
            Incoming_link.Jump_inst = 'Jump to' Target_address;
        ENDTHEN;
    ENDTHEN;

    ELSE BEGIN

       /* Snap a link for external data. */

        Outgoing_link (0) = 'Load pointer' Target_address;
        Outgoing_link (1) = 'Return instruction';
    ENDELSE;

END Snap_the_Links;
```

```
EXECUTIVE Address Space Manager;

/*
     Explanation of variables :

     PRT_size - The size of the process reference table.

     Seg_number  -  The  segment  number assigned to a newly
       loaded object by the procedure Load_Object.

     PRT - The process reference table.


     /* end of variable explanation */




/*  The following is a list of variable and constant
    declaration used in the Address Space Manager. */

DECLARE

   PRT_size : INTEGER;
   Seg_Number : ADDRESS;

   PRT : ARRAY of STRUCTURES of
         Valid_bit : BOOLEAN;
         Name : ARRAY of CHARACTERS;
         Seg_number : ADDRESS;
         END;

END of DECLARATIONS;
```

```
/***********************************************************

ASM_Make_Accessable performs the following functions:

 1.  Determines if the object passed as an arguments is
     already in the process reference table (i.e., is
     already in the process address space).

 2.  If not, loads the object into memory at the next
     available memory location and updates the Process
     Reference Table (PRT).

 3.  Returns to the point of call the segment number
     assigned to the object.

***********************************************************/

PROCEDURE ASM_Make_Accessable;

  PARAMETER_LIST : Object_name;

    DECLARE i : INTEGER;
            found : BOOLEAN;

    i = 1;
    found = false;

    /* check to see if Object_name is in the PRT */

     DO WHILE NOT found AND i <= PRT_size;

       IF PRT(i).valid_bit = valid THEN BEGIN
           IF PRT(i).name = Object_name
             THEN found = true;
           ELSE i = i + 1;
       ENDTHEN;

     ENDWHILE;
```

```
      IF NOT found THEN BEGIN

        /* find a free PRT entry */

        i = 1;
        DO WHILE PRT(i).valid_bit = valid;
            i = i + 1;
        ENDWHILE;

          CALL Load_object;
            PARAMETER_LIST : Object_name;
            RETURN_LIST    : Seg_Number;

          PRT(i).name = Object_Name;
          PRT(i).seg_number = Seg_Number;
          PRT(i).valid_bit = valid;

       ENDTHEN;

     RETURN_LIST : PRT(i).seg_number;

 END ASM_Make_Accessable;
```

```
/****************************************************************

ASM_Remove_Seg performs the following functions :

 1.  Removes an object from a process address space.

****************************************************************/

PROCEDURE ASM_Remove_Seg;

  PARAMETER_LIST : Object_name;

    DECLARE i : INTEGER;
            found : BOOLEAN;

    /* find Object_name in the process reference table (PRT) */

    i = 1;
    found = false;

    DO WHILE NOT found AND i <= PRT_size;

        IF PRT(i).name = Object_name
          THEN found = true;
        ELSE i = i + 1;

    ENDWHILE;

    /* remove the object from the PRT */

    PRT(i).valid_bit = invalid;

   RETURN_LIST : none;

 END ASM_Remove_seg;
```

APPENDIX B - DEMONSTRATION SOURCE LISTINGS

```
PL/M-80 COMPILER    PROCESS INITIALIZATION

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE EXEC
OBJECT MODULE PLACED IN :F1:EXEC.OBJ
COMPILER INVOKED BY:  PLM80 :F1:EXEC.SRC PAGELENGTH(38) TITLE('PROCESS INITIALIZATION')


   1          EXEC : DO;


              /* DATE LAST EDITED : 29 JULY 1980 */

              /***     P R O C E S S   I N I T I A L I Z A T I O N     ***/


              /* LITERALS */

   2   1      DECLARE LIT LITERALLY 'LITERALLY',
                      POINTER LIT 'ADDRESS',
                      INTEGER LIT 'ADDRESS',
                      TRUE LIT '01H',
                      FALSE LIT '00H',
                      SPACE LIT '20H',

                      POP$H LIT '0E1H',
                      POP$D LIT '0D1H',
                      PCHL  LIT '0E9H',
                      CALL$INST LIT '0CDH',
                      RETURN$INST LIT '0C9H',
                      LXI$B LIT '01H',

                      SET LIT '01H',
                      NOT$SET LIT '00H';

               /*** PROGRAM VARIABLES ***/
```

```
 3   1      DECLARE RET$VALUE$PTR POINTER,
                    OBJECT STRUCTURE (
                      UNIQUE$ID BYTE,
                      BASE$ADDRESS ADDRESS),

                    LINKER$VALUES$PTR POINTER,
                    LINKER$VALUES STRUCTURE (
                      LINKER$ADDRESS ADDRESS,
                      LINK$ADDR$TABLE$BASE ADDRESS,
                      LINK$TABLE$ADDRESS ADDRESS),

                    PROGRAM$POINTER POINTER,
                    PROGRAM STRUCTURE (
                      NAME (12) BYTE,
                      SIZE BYTE),

                    DISPLAY$TOGGLE BYTE,
                    TYPE$PROCEDURE BYTE INITIAL (00H);


            /***********************************************************************
             *                                                                     *
             *                EXTERNAL PROCEDURE DECLARATIONS                      *
             *                                                                     *
             ***********************************************************************/

             /*** DISPLAY OUTPUTS AN ASCII CHARACTER STRING TO THE CRT. ***/

 4   1       DISPLAY : PROCEDURE (STRING$ADDRESS) EXTERNAL;
 5   2         DECLARE STRING$ADDRESS POINTER;
 6   2       END DISPLAY;

             /*** INITIALIZE$ASM INITIALIZES THE ADDRESS SPACE MANAGER ***/

 7   1       INITIALIZE$ASM : PROCEDURE EXTERNAL;
```

```
  8   2          END INITIALIZE$ASM;

                 /*** ASM$MAKE$ACCESSABLE ENTERS AN OBJECT IN THE ADDRESS SPACE
                      OF THE EXECUTING PROCESS. ***/

  9   1          ASM$MAKE$ACCESSABLE : PROCEDURE (OBJ$NAME$PTR, RETURN$VALUE$PTR)
                                                          EXTERNAL;

 10   2            DECLARE OBJ$NAME$PTR POINTER,
                           RETURN$VALUE$PTR POINTER;
 11   2          END ASM$MAKE$ACCESSABLE;

                 /*** DISPLAY$PRT DISPLAYS THE PROCESS REFERENCE TABLE ON THE CRT ***/

 12   1          DISPLAY$PRT : PROCEDURE EXTERNAL;
 13   2          END DISPLAY$PRT;

                 /*** OUTPUT$THE$LINK$TABLE DISPLAYS THE COMBINED LINKAGE TABLE ON
                      THE CRT. ***/

 14   1          OUTPUT$THE$LINK$TABLE : PROCEDURE (LINK$ADDRESS$TABLE$BASE) EXTERNAL;
 15   2            DECLARE LINK$ADDRESS$TABLE$BASE ADDRESS;
 16   2          END OUTPUT$THE$LINK$TABLE;

                 /*** INITIALIZE$LINKER INITIALIZES THE LINKER AND RETURNS THE ADDRESS
                      OF THE CONTROL MODULE "LINKER" AND THE BASE ADDRESS OF THE
                      LINKAGE ADDRESS TABLE. ***/

 17   1          INITIALIZE$LINKER : PROCEDURE (RETURN$VALUES$POINTER) EXTERNAL;
 18   2            DECLARE RETURN$VALUES$POINTER POINTER;
 19   2          END INITIALIZE$LINKER;

                 /*** LINKAGE$TABLE$ROUTINES BUILDS A LINKAGE TABLE FOR AN OBJECT
                      AND ENTERS THE OBJECT'S LINKAGE TABLE ADDRESS IN THE LINKAGE
                      ADDRESS TABLE.  ***/
```

```
 20   1       LINKAGE$TABLE$ROUTINES : PROCEDURE (OBJECT$SEG$NUMBER,
                                                  OBJECT$BASE$ADDRESS,
                                                  OBJECT$TYPE,
                                                  POINTER$TO$SYMBOLIC$NAME)
                                       EXTERNAL;

 21   2         DECLARE OBJECT$SEG$NUMBER BYTE,
                        OBJECT$BASE$ADDRESS ADDRESS,
                        OBJECT$TYPE BYTE,
                        POINTER$TO$SYMBOLIC$NAME POINTER;

 22   2       END LINKAGE$TABLE$ROUTINES;

              /*** BOOT RETURNS CONTROL TO THE OPERATING SYSTEM ***/

 23   1       BOOT : PROCEDURE EXTERNAL;
 24   2       END BOOT;

              /*** CRLF OUTPUTS A CARRIAGE RETURN AND LINE FEED ON THE CRT. ***/

 25   1       CRLF : PROCEDURE EXTERNAL;
 26   2       END CRLF;


              /**************************************************************/

              /***               PROCESS INITIALIZATION ROUTINES          ***/

              /**************************************************************/

              /***************************************************************
               *                                                             *
               *  READ$COMMAND$LINE READS THE NAME OF THE PROGRAM TO BE      *
               *    EXECUTED AND SETS THE DISPLAY TOGGLE.                    *
```

```
                 *                                                               *
                 ****************************************************************/

 27   1          READ$COMMAND$LINE : PROCEDURE (NAME$POINTER);

 28   2            DECLARE NAME$POINTER POINTER,
                           OBJECT BASED NAME$POINTER STRUCTURE (
                             NAME (12) BYTE,
                             SIZE BYTE),

                           I BYTE,
                           INPUT$POINTER POINTER,
                           INPUT$BUFFER BASED INPUT$POINTER (12) BYTE;

 29   2            I = 0;

                   /*** THE CP/M OPERATING SYSTEM STORES THE COMMAND LINE IN A BUFFER
                        STARTING AT 80H.  THE BYTE AT 80H CONTAINS THE BUFFER SIZE
                        WHILE STARTING AT 82H IS THE ACTUAL COMMAND LINE.  THUS, TO
                        RUN A PROGRAM, THE FOLLOWING COMMAND LINE IS INPUTED:

                              A> EXEC PROGRAM $

                        WHERE 'A>' IS THE CP/M PROMPT; EXEC IS THE DYNAMIC LINKER
                        ROUTINE; 'PROGRAM' IS THE PROGRAM NAME; AND '$' INDICATES
                        WHETHER THE LINKAGE TABLE AND PROCESS REFERENCE TABLE ARE TO
                        BE DISPLAYED OR NOT.  IN THIS CASE,THE COMMAND LINE IS:

                             'PROGRAM $'.     ***/

 30   2            INPUT$POINTER = 82H;

                   /*** COPY THE NAME OF THE PROGRAM TO BE EXECUTED INTO THE NAME
                        BUFFER. ***/
```

```
 31   2          DO WHILE INPUT$BUFFER (I) <> SPACE;
 32   3             OBJECT.NAME(I) = INPUT$BUFFER (I);
 33   3             I = I + 1;
 34   3          END;

                 /*** SET THE SIZE OF THE OBJECT NAME ***/

 35   2          OBJECT.SIZE = I;

                 /*** SET THE OBJECT TYPE TO EXECUTABLE CODE (TYPE "COM"). ***/

 36   2          OBJECT.NAME (I)     = '.';
 37   2          OBJECT.NAME (I + 1) = 'C';
 38   2          OBJECT.NAME (I + 2) = 'O';
 39   2          OBJECT.NAME (I + 3) = 'M';

                 /*** NOW SEE IF THE DISPLAY TOGGLE SHOULD BE SET ***/

 40   2          IF INPUT$BUFFER (I + 1) = '$' THEN DISPLAY$TOGGLE = SET;
 42   2          ELSE DISPLAY$TOGGLE = NOT$SET;

 43   2       END READ$COMMAND$LINE;

             /************************************************************************
              *                                                                      *
              *  EXECUTE IS THE KEY TO INVOKING THE PROGRAM TO BE EXECUTED           *
              *  AND SETTING THE LINKAGE POINTER.   IT LOADS THE LINKAGE             *
              *  POINTER IN THE B & C REGISTER PAIR (THE DESIGNATED LINKAGE          *
              *  POINTER REGISTER) AND THEN INVOKES THE PROGRAM TO BE                *
              *  EXECUTED.  IT DOES THIS BY INITIAIIZING AN ARRAY WITH THE           *
              *  MACHINE INSTRUCTIONS REQUIRED AND THEN EXECUTING THE                *
              *  ARRAY.                                                              *
              *                                                                      *
              ************************************************************************/
```

```
 44   1         EXECUTE : PROCEDURE (LINKAGE$POINTER, OBJECT$ADDRESS);

 45   2            DECLARE LINKAGE$POINTER POINTER,
                           OBJECT$ADDRESS ADDRESS,

                           EXECUTE$ARRAY$BASE ADDRESS,
                           EXECUTE$ARRAY STRUCTURE (
                             BYTE1 BYTE,
                             BYTE2$3 ADDRESS,
                             BYTE4 BYTE,
                             BYTE5$6 ADDRESS,
                             BYTE7 BYTE);

               /*** SET EXECUTE$ARRAY$BASE TO POINT TO EXECUTE$ARRAY. ***/

 46   2        EXECUTE$ARRAY$BASE = .EXECUTE$ARRAY.BYTE1;

 47   2        EXECUTE$ARRAY.BYTE1   =  LXI$B;
 48   2        EXECUTE$ARRAY.BYTE2$3 =  LINKAGE$POINTER;
 49   2        EXECUTE$ARRAY.BYTE4   =  CALL$INST;
 50   2        EXECUTE$ARRAY.BYTE5$6 =  OBJECT$ADDRESS;
 51   2        EXECUTE$ARRAY.BYTE7   =  RETURN$INST;

               /*** NOW EXECUTE EXECUTE$ARRAY ***/

 52   2        CALL EXECUTE$ARRAY$BASE;

 53   2     END EXECUTE;

           /*****************************************************************
            *                                                               *
            *  BUILD$INTERRUPT$VECTOR INITIALIZES THE INTERRUPT VECTOR      *
            *  TO CALL THE LINKER AND THEN JUMP TO THE OUTGOING LINK.       *
            *                                                               *
            *****************************************************************/
```

```
 54   1        BUILD$INTERRUPT$VECTOR : PROCEDURE (LINKER$ADDRESS);

 55   2          DECLARE INTERRUPT$BASE POINTER,
                         INTERRUPT$VECTOR BASED INTERRUPT$BASE STRUCTURE (
                           BYTE1 BYTE,
                           BYTE2 BYTE,
                           BYTES3$4 ADDRESS,
                           BYTE5 BYTE,
                           BYTE6 BYTE),

                         LINKER$ADDRESS ADDRESS;


                /*** THE INTERRUPT VECTOR INVOKES THE LINKER VIA AN INTERRUPT
                     GENERATED BY THE INITIALIZED OUTGOING LINK.  THE INSTRUCTION
                     IN THE OUTGOING LINK WHICH CALLS THE INTERRUPT VECTOR IS A
                     RESET 4 INSTRUCTION (RST 4).  THIS INSTRUCTION SAVES THE
                     RETURN ADDRESS ON THE STACK AND JUMPS TO LOCATION 20H.  THE
                     INTERRUPT VECTOR REMOVES THE RETURN ADDRESS FROM THE STACK
                     (VIA THE POP$H INSTRUCTION) AND CALL THE LINKER.  WHEN THE
                     LINKER HAS FINISHED EXECUTING IT RETURNS THE BASE ADDRESS OF
                     THE (SNAPPED) OUTGOING LINK TO THE INTERRUPT VECTOR.  THE
                     INTERRUPT VECTOR RESTORES THE PARAMETER REGISTER (POP$D) AND
                     JUMPS TO THE OUTGOING LINK (PCHL). ***/

 56   2         INTERRUPT$BASE = 20H;

 57   2         INTERRUPT$VECTOR.BYTE1     = POP$H;
 58   2         INTERRUPT$VECTOR.BYTE2     = CALL$INST;
 59   2         INTERRUPT$VECTOR.BYTES3$4  = LINKER$ADDRESS;
 60   2         INTERRUPT$VECTOR.BYTE5     = POP$D;
 61   2         INTERRUPT$VECTOR.BYTE6     = PCHL;

 62   2        END BUILD$INTERRUPT$VECTOR;
```

```
                 /**********************************************************************/

                 /****                      M A I N   C O D E                      ****/

                 /**********************************************************************/


  63   1         CALL CRLF;
  64   1         CALL DISPLAY (.('DYNAMIC LINKER VERSION 1.0', '$'));
  65   1         CALL CRLF;
  66   1         CALL CRLF;

  67   1         LINKER$VALUES$PTR = .LINKER$VALUES.LINKER$ADDRESS;
  68   1         PROGRAM$POINTER = .PROGRAM.NAME(0);
  69   1         RET$VALUE$PTR = .OBJECT.UNIQUE$ID;

  70   1         CALL READ$COMMAND$LINE (PROGRAM$POINTER);

  71   1         CALL INITIALIZE$ASM;
  72   1         CALL INITIALIZE$LINKER (LINKER$VALUES$PTR);

  73   1         CALL ASM$MAKE$ACCESSABLE (PROGRAM$POINTER, RET$VALUE$PTR);

  74   1         CALL LINKAGE$TABLE$ROUTINES (OBJECT.UNIQUE$ID,
                                              OBJECT.BASE$ADDRESS,
                                              TYPE$PROCEDURE,
                                              PROGRAM$POINTER);

  75   1         CALL BUILD$INTERRUPT$VECTOR (LINKER$VALUES.LINKER$ADDRESS);
  76   1         CALL EXECUTE (LINKER$VALUES.LINK$TABLE$ADDRESS, OBJECT.BASE$ADDRESS);

  77   1         IF DISPLAY$TOGGLE = SET THEN DO;
  79   2            CALL DISPLAY$PRT;
  80   2            CALL CRLF;
```

```
  81   2             CALL OUTPUT$THE$LINK$TABLE (LINKER$VALUES.LINK$ADDR$TABLE$BASE);
  82   2          END;

  83   1          CALL BOOT;

  84   1        END EXEC;


MODULE INFORMATION:

     CODE AREA SIZE      = 01B2H    434D
     VARIABLE AREA SIZE  = 0034H     52D
     MAXIMUM STACK SIZE  = 0006H      6D
     315 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

```
ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE DLKR
OBJECT MODULE PLACED IN :F1:DLKR.OBJ
COMPILER INVOKED BY:  PLM80 :F1:DLKR.SRC PAGELENGTH(38) TITLE('LINKER MODULE')


   1            DLKR : DO;

                /* DATE LAST EDITED : 4 AUGUST 1980 */

   2   1        DECLARE LIT LITERALLY 'LITERALLY',
                        TRUE LIT '01H',
                        FALSE LIT '00H',
                        SPACE LIT '20H',

                        BOOLEAN LIT 'BYTE',
                        FUNCTION LIT 'PROCEDURE',
                        POINTER LIT 'ADDRESS',
                        INTEGER LIT 'ADDRESS',
                        AN$ENTRY$POINT LIT 'PUBLIC',

                        LOAD$LP$INST LIT '01H',
                        LOAD$POINTER LIT '11H',
                        JUMP$TO LIT '0C3H',
                        RETURN$INST LIT '0C9H',

                        UNSNAPPED LIT '00H',
                        VALID LIT '01H',
                        INVALID LIT '00H';


                /*** VARIABLE DECLARATIONS ***/

   3   1        DECLARE
```

```
EN$BUFFER STRUCTURE (
   NAME (16) BYTE,
   SIZE BYTE),
EN$BUFFER$PTR POINTER,

FIXED$SN$OFFSET BYTE INITIAL (05H),
FREE$LINK$TABLE ADDRESS,

IN$LINK$ADDRESS POINTER,
INCOMING$LINK BASED IN$LINK$ADDRESS STRUCTURE (
   LOAD$LP (3) BYTE,
   JUMP$INST (3) BYTE),

LINKAGE$TABLE (1024) BYTE,

LINKAGE$POINTER POINTER,
LINKAGE$ARRAY BASED LINKAGE$POINTER STRUCTURE (
   SIZE INTEGER,
   SNT$ADDRESS ADDRESS,
   BODY (1) BYTE),

LINKAGE$ADDRESS$TABLE (16) STRUCTURE (
   VALID$BIT BYTE,
   BASE$ADDR ADDRESS),

NEW$LINK$PTR POINTER,
NEW$LINK$TABLE BASED NEW$LINK$PTR STRUCTURE (
   SIZE INTEGER,
   SNT$ADDRESS ADDRESS,
   BODY (1) BYTE),

OBJECT STRUCTURE (
   UNIQUE$ID BYTE,
   BASE$ADDRESS ADDRESS),
OBJECT$ID$POINTER POINTER,
```

```
OBJECT$TYPE BYTE,

OUT$LINK$ADDRESS POINTER,
OUTGOING$LINK BASED OUT$LINK$ADDRESS (4) BYTE,

SN$BUFFER STRUCTURE (
   NAME (12) BYTE,
   SIZE BYTE),
SN$BUFFER$POINTER POINTER,

SN$ADDRESS POINTER,
SN$ITEM BASED SN$ADDRESS STRUCTURE (
   DESCRIPTOR BYTE,
   LINK$OFFSET INTEGER,
   ENTRY$POINT INTEGER,
   NAME (1) BYTE),

SN$OFFSET INTEGER,
SN$SIZE BYTE,
SN$SIZE$MASK BYTE INITIAL (1FH),
SN$TYPE$MASK BYTE INITIAL (80H),

TARGET$ADDRESS ADDRESS,

TEMPLATE$BASE$ADDRESS POINTER,
TEMPLATE BASED TEMPLATE$BASE$ADDRESS STRUCTURE (
   SIZE INTEGER,
   SNT$OFFSET INTEGER,
   BODY (1) BYTE),

TYPE$DATA BYTE INITIAL (01H),
TYPE$PROCEDURE BYTE INITIAL (00H);
```

```
                  /*** END OF VARIABLE DECLARATIONS ***/


                  /***************************************************************
                   *                                                             *
                   *      EXTERNALLY DEFINED SYSTEM PROCEDURE DECLARATIONS       *
                   *                                                             *
                   **************************************************************/

                   /*** DISPLAY OUTPUTS AN ASCII CHARACTER STRING TO THE CRT. ***/

   4   1           DISPLAY : PROCEDURE (STRING$ADDRESS) EXTERNAL;
   5   2             DECLARE STRING$ADDRESS POINTER;
   6   2           END DISPLAY;

                   /*** OUTPUT$ADDR DISPLAYS A 2-BYTE VALUE ON THE CRT. ***/

   7   1           OUTPUT$ADDR : PROCEDURE (DEVICE, VALUE) EXTERNAL;
   8   2             DECLARE VALUE ADDRESS,
                             DEVICE BYTE;
   9   2           END OUTPUT$ADDR;

                   /*** DISPLAY$CHAR OUTPUTS AN ASCII CHARACTER TO THE CRT. ***/

  10   1           DISPLAY$CHAR : PROCEDURE (CHARACTER) EXTERNAL;
  11   2             DECLARE CHARACTER BYTE;
  12   2           END DISPLAY$CHAR;

                   /*** CRLF GENERATES A CARRIAGE RETURN AND LINE FEED ON THE CRT. ***/

  13   1           CRLF : PROCEDURE EXTERNAL;
  14   2           END CRLF;

                   /*** END OF EXTERNAL SYSTEM DECLARATIONS. ***/
```

```
                 /***   ADDRESS SPACE MANAGER EXTERNAI ROUTINE DECLARATIONS   ***/

  15   1         ASM$MAKE$ACCESSABLE : PROCEDURE (OBJ$NAME$PTR, RETURN$VALUE$PTR)
                                              EXTERNAL;

  16   2           DECLARE OBJ$NAME$PTR POINTER,
                           RETURN$VALUE$PTR POINTER;
  17   2         END ASM$MAKE$ACCESSABLE;

  18   1         ASM$REMOVE$SEG : PROCEDURE (OBJ$NAME$PTR) EXTERNAL;
  19   2           DECLARE OBJ$NAME$PTR POINTER;
  20   2         END ASM$REMOVE$SEG;


                 /***  END OF ADDRESS SPACE MANAGER EXTERNAL DECLARATIONS ***/


                 /*********************************************************************/

                 /*                T H E   D Y N A M I C   L I N K E R               */

                 /*********************************************************************/

                 /*********************************************************************
                  *                                                                   *
                  *   ACCESS$SYMBOLIC$NAME$DATA PERFORMS THE FOLLOWING FUNCTIONS:     *
                  *                                                                   *
                  *   1.  OBTAINS THE ADDRESS OF THE SYMBOLIC NAME OF THE EXTERNAL    *
                  *       REFERENCE BEING LINKED.                                     *
                  *                                                                   *
                  *   2.  LOADS THE SYMBOLIC NAME  OF THE EXTERNAL REFERENCE  IN      *
                  *       THE SYMBOLIC NAME BUFFER (SN$BUFFER).                       *
                  *                                                                   *
                  *   3.  LOADS THE ENTRY NAME IN THE EXTERNAL REFERENCE IN THE       *
```

```
                     *      ENTRY NAME BUFFER (EN$BUFFER).                            *
                     *                                                                *
                     *   4.  COMPUTES THE OUTGOING LINK ADDRESS AND DETERMINES        *
                     *       WHETHER THE EXTERNAL OBJECT IS A PROCEDURE OR DATA.      *
                     *                                                                *
                     ******************************************************************/

  21   1             ACCESS$SYMBOLIC$NAME$DATA : PROCEDURE (LINKAGE$POINTER, SN$OFFSET);

  22   2               DECLARE LINKAGE$POINTER POINTER,
                               SN$OFFSET INTEGER,
                               (I, TEMP) BYTE;

  23   2               SN$ADDRESS = LINKAGE$ARRAY.SNT$ADDRESS + SN$OFFSET;
  24   2               SN$SIZE = SN$ITEM.DESCRIPTOR AND SN$SIZE$MASK;

                       /*** LOAD THE SYMBOLIC NAME INTO SN$BUFFER.NAME ***/

  25   2               I = 0;
  26   2               DO WHILE (SN$ITEM.NAME(I) <> ':') AND
                                (I < SN$SIZE);

  27   3                 SN$BUFFER.NAME(I) = SN$ITEM.NAME(I);
  28   3                 I = I + 1;

  29   3               END;   /* OF THE WHILE CLAUSE */

                       /*** LOAD THE SYMBOLIC NAME SIZE INTO SN$BUFFER.SIZE ***/

  30   2               SN$BUFFER.SIZE = I;

                       /*** LOAD THE ENTRY NAME BUFFER WITH THE ENTRY NAME ***/

  31   2               IF I = SN$SIZE THEN DO;
```

```
                        /* NO ENTRY NAME SPECIFIED, DEFAULT TO THE SYMBOLIC
                           NAME AS THE ENTRY POINT */

 33   3                 EN$BUFFER.SIZE = SN$SIZE;
 34   3                 DO I = 0 TO (SN$SIZE - 1);
 35   4                    EN$BUFFER.NAME(I) = SN$ITEM.NAME(I);
 36   4                 END;

 37   3              END;   /* OF THE THEN CLAUSE */

 38   2              ELSE DO;

 39   3                TEMP = I;
 40   3                I = I + 1;

                       /* LOAD SIZE OF ENTRY NAME INTO EN$BUFFER.SIZE */

 41   3                EN$BUFFER.SIZE = SN$SIZE - I;

                       /* LOAD ENTRY NAME INTO ENTRY NAME BUFFER */

 42   3                DO WHILE (I < SN$SIZE);
 43   4                   EN$BUFFER.NAME(I - TEMP - 1) = SN$ITEM.NAME(I);
 44   4                   I = I + 1;
 45   4                END;

 46   3              END;    /* OF THE ELSE CLAUSE */

                     /*** COMPUTE THE ADDRESS OF THE OUTGOING LINK AND DETERMINE
                          THE TYPE (PROCEDURE OR DATA) OF THE EXTERNAL OBJECT. ***/

 47   2              OUT$LINK$ADDRESS = LINKAGE$POINTER + SN$ITEM.LINK$OFFSET;

 48   2              IF (SN$ITEM.DESCRIPTOR AND SN$TYPE$MASK) = 0 THEN
 49   2                OBJECT$TYPE = TYPE$PROCEDURE;
```

```
  50   2              ELSE OBJECT$TYPE = TYPE$DATA;

                      /*** THE CP/M OPERATING SYSTEM UTILIZES A FILETYPE OF 'COM' FOR
                           PROCEDURES AND 'DTA' FOR DATA.  NOW THAT WE KNOW THE OBJECT
                           TYPE WE WILL INSERT THE FILETYPE IN SN$BUFFER.NAME
                           ACCORDINGLY.  ***/

  51   2              IF OBJECT$TYPE = TYPE$PROCEDURE THEN DO;

  53   3                   I = SN$BUFFER.SIZE;

  54   3                   SN$BUFFER.NAME(I)     = '.';
  55   3                   SN$BUFFER.NAME(I + 1) = 'C';
  56   3                   SN$BUFFER.NAME(I + 2) = 'O';
  57   3                   SN$BUFFER.NAME(I + 3) = 'M';
  58   3              END;
  59   2              ELSE DO;

                           /* THE OBJECT IS TYPE DATA */

  60   3                   I = SN$BUFFER.SIZE;

  61   3                   SN$BUFFER.NAME(I)     = '.';
  62   3                   SN$BUFFER.NAME(I + 1) = 'D';
  63   3                   SN$BUFFER.NAME(I + 2) = 'T';
  64   3                   SN$BUFFER.NAME(I + 3) = 'A';

  65   3              END;

  66   2           END ACCESS$SYMBOLIC$NAME$DATA;

                   /***************************************************************
                    *                                                              *
                    *   BUILD$OBJECT$LINK PERFORMS THE FOLLOWING FUNCTIONS:        *
```

```
                  *                                                        *
                  *   1.  CAUSES THE ADDRESS SPACE MANAGER TO LOAD THE EXTERNAL *
                  *       OBJECT'S LINKAGE TABLE TEMPLATE INTO THE PROCESS      *
                  *       ADDRESS SPACE.                                   *
                  *                                                        *
                  *   2.  INITIALIZES A TEMPORARY VARIABLE (NEW$LINK$PTR) TO    *
                  *       THE VALUE OF THE OBJECT'S LINKAGE POINTER.       *
                  *                                                        *
                  *   3.  APPENDS OBJECT.LINK TO THE END OF THE COMBINED   *
                  *       LINKAGE TABLE.                                   *
                  *                                                        *
                  *   4.  DELETES THE LINKAGE TABLE TEMPLATE FROM THE PROCESS   *
                  *       ADDRESS SPACE.                                   *
                  *                                                        *
                  **********************************************************/

 67   1           BUILD$OBJECT$LINK : PROCEDURE (BASE$ADDRESS, OBJECT$TYPE,
                                                 SN$POINTER);

 68   2             DECLARE RETURN$VALUE$PTR POINTER,
                            RETURN$VALUE STRUCTURE (
                              UNIQUE$ID BYTE,
                              BASE$ADDR ADDRESS),
                            BASE$ADDRESS ADDRESS,
                            OBJECT$TYPE BYTE,
                            SN$POINTER POINTER,
                            SYMBOLIC$NAME BASED SN$POINTER STRUCTURE (
                               NAME (12) BYTE,
                               SIZE BYTE),
                            I BYTE;

                    /*** INITIALIZE RETURN$VALUE$PTR ***/

 69   2             RETURN$VALUE$PTR = .RETURN$VALUE.UNIQUE$ID;
```

```
                    /*** APPEND A FILE TYPE OF TEMPLATE TO THE SYMBOLIC NAME ***/

 70   2             I = SYMBOLIC$NAME.SIZE;

 71   2             SYMBOLIC$NAME.NAME(I:=I + 1) = 'T';
 72   2             SYMBOLIC$NAME.NAME(I:=I + 1) = 'M';
 73   2             SYMBOLIC$NAME.NAME(I:=I + 1) = 'P';

 74   2             CALL ASM$MAKE$ACCESSABLE (SN$POINTER, RETURN$VALUE$PTR);

                    /*** SET THE TEMPLATE BASE ADDRESS = BASE$ADDR OF THE TEMPLATE
                         AND, NEW$LINK$PTR = FREE$LINK$TABLE ***/

 75   2             TEMPLATE$BASE$ADDRESS = RETURN$VALUE.BASE$ADDR;
 76   2             NEW$LINK$PTR = FREE$LINK$TABLE;

 77   2             IF OBJECT$TYPE = TYPE$PROCEDURE THEN DO;

                      /* IF THE OBJECT IS A PROCEDURE, THEN ITS SYMBOLIC NAME
                         TABLE IS IN THE OBJECT CODE SEGMENT. */

 79   3                NEW$LINK$TABLE.SNT$ADDRESS = TEMPLATE.SNT$OFFSET +
                                                    BASE$ADDRESS;

 80   3             END;   /* OF THE THEN CLAUSE */

 81   2             ELSE DO;

                      /* THE OBJECT IS DATA AND ITS SYMBOLIC NAME TABLE IS IN
                         THE TEMPLATE */

 82   3               NEW$LINK$TABLE.SNT$ADDRESS = NEW$LINK$PTR + TEMPLATE.SNT$OFFSET;

 83   3             END;   /* OF THE ELSE CLAUSE */
```

```
                      /*** NOW BUILD THE REST OF THE LINKAGE TABLE ***/

 84   2               NEW$LINK$TABLE.SIZE = TEMPLATE.SIZE;

 85   2               DO I = 0 TO (TEMPLATE.SIZE - 5);
 86   3                  NEW$LINK$TABLE.BODY (I) = TEMPLATE.BODY(I);
 87   3               END;

 88   2               FREE$LINK$TABLE = NEW$LINK$PTR + NEW$LINK$TABLE.SIZE + 1;

 89   2               CALL ASM$REMOVE$SEG (SN$POINTER);

 90   2            END BUILD$OBJECT$LINK;

            /*****************************************************************
             *                                                               *
             *   LINKAGE$TABLE$ROUTINES PERFORMS THE FOLLOWING FUNCTIONS:    *
             *                                                               *
             *   1.  DETERMINES IF A LINKAGE TABLE ALREADY EXIST FOR THE     *
             *       EXTERNAL REFERENCE BEING LINKED.                        *
             *                                                               *
             *       A.  IF NOT, LINKAGE$TABLE$ROUTINES INITIALIZES THE      *
             *       LINKAGE ADDRESS TABLE ENTRY FOR THE OBJECT AND THEN     *
             *       CALLS ON BUILD$OBJECT$LINK.                             *
             *                                                               *
             *       B.  IF SO, LINKAGE$TABLE$ROUTINES SETS A TEMPORARY      *
             *       VARIABLE (NEW$LINK$PTR) EQUAL TO THE LINKAGE POINTER    *
             *       VALUE FOR THE NEW OBJECT'S LINKAGE TABLE.               *
             *                                                               *
             *****************************************************************/

 91   1     LINKAGE$TABLE$ROUTINES : PROCEDURE (OBJECT$SEG$NUMBER, BASE$ADDR,
                                                  OBJECT$TYPE, SN$POINTER)
               /* LINKAGE$TABLE$ROUTINES IS */   AN$ENTRY$POINT;
```

```
 92   2          DECLARE OBJECT$SEG$NUMBER BYTE,
                         BASE$ADDR ADDRESS,
                         OBJECT$TYPE BYTE,
                         SN$POINTER POINTER;

 93   2          IF LINKAGE$ADDRESS$TABLE (OBJECT$SEG$NUMBER).VALID$BIT <> VALID
                 THEN DO;

                   /*** THIS IS THE FIRST TIME THE OBJECT HAS BEEN REFERENCED
                        BY THE PROCESS AND THE LINKER MUST BUILD A LINKAGE TABLE
                        FOR THE OBJECT. ***/

 95   3            LINKAGE$ADDRESS$TABLE (OBJECT$SEG$NUMBER).BASE$ADDR =
                                          FREE$LINK$TABLE;
 96   3            LINKAGE$ADDRESS$TABLE (OBJECT$SEG$NUMBER).VALID$BIT = VALID;

 97   3            CALL BUILD$OBJECT$LINK (BASE$ADDR, OBJECT$TYPE, SN$POINTER);

 98   3          END;   /* OF THE THEN CLAUSE */

                 ELSE

                   /*** THE OBJECT ALREADY HAS A LINKAGE TABLE ***/

 99   2            NEW$LINK$PTR =
                        LINKAGE$ADDRESS$TABLE (OBJECT$SEG$NUMBER).BASE$ADDR;

100   2        END LINKAGE$TABLE$ROUTINES;

              /*****************************************************************
               *                                                               *
               *    ACCESS$ENTRY$NAME$DATA PERFORMS THE FOLLOWING FUNCTIONS:  *
               *                                                               *
               *    1.  COMPUTES THE ADDRESS (TARGET$ADDRESS) IN THE EXTERNAL *
               *        OBJECT TO BE UTILIZED IN THE LINKING PROCESS.         *
```

```
                  *                                                        *
                  *   2.  COMPUTES THE INCOMING LINK ADDRESS (IF APPLICABLE).   *
                  *                                                        *
                  **********************************************************/

 101   1          ACCESS$ENTRY$NAME$DATA : PROCEDURE;

 102   2             DECLARE I BYTE,
                             FOUND BOOLEAN;

                    /*** GET$NEXT$SN$ITEM STEPS THROUGH A SYMBOLIC NAME TABLE
                         AN ENTRY AT A TIME. ***/

 103   2            GET$NEXT$SN$ITEM : PROCEDURE;
 104   3                SN$ADDRESS = SN$ADDRESS + FIXED$SN$OFFSET +
                                     (SN$ITEM.DESCRIPTOR AND SN$SIZE$MASK);
 105   3            END GET$NEXT$SN$ITEM;

                    /* . . . . . . . . . . . . . . . . . . . . . . . . . . . */

                    /*** NAMES$MATCH CHECKS A SYMBOLIC NAME TABLE ENTRY AND
                         EN$BUFFER.NAME FOR A MATCH.  ***/

 106   2            NAMES$MATCH : FUNCTION BOOLEAN;
 107   3                 DECLARE I BYTE,
                                 RESULT BOOLEAN;

 108   3                 RESULT = TRUE;
 109   3                 I = 0;

 110   3                 DO WHILE I < EN$BUFFER.SIZE AND RESULT = TRUE;
 111   4                    IF EN$BUFFER.NAME(I) <> SN$ITEM.NAME(I) THEN
 112   4                       RESULT = FALSE;
 113   4                    ELSE I = I + 1;
 114   4                 END;  /* OF THE WHILE LOOP */
```

```
115   3                  RETURN RESULT;

116   3              END NAMES$MATCH;

                     /* . . . . . . . . . . . . . . . . . . . . . . . . . . . . */


                     /*** BEGIN ACCESS$ENTRY$NAME$DATA ***/

117   2              FOUND = FALSE;
118   2              SN$ADDRESS = NEW$LINK$TABLE.SNT$ADDRESS;

119   2              DO WHILE NOT FOUND;
120   3                 IF NAMES$MATCH THEN FOUND = TRUE;
122   3                 ELSE CALL GET$NEXT$SN$ITEM;
123   3              END;

124   2              TARGET$ADDRESS = OBJECT.BASE$ADDRESS + SN$ITEM.ENTRY$POINT;

125   2              IF OBJECT$TYPE = TYPE$PROCEDURE THEN
126   2                IN$LINK$ADDRESS = NEW$LINK$PTR + SN$ITEM.LINK$OFFSET;

127   2          END ACCESS$ENTRY$NAME$DATA;

               /*****************************************************************
                *                                                               *
                *   SNAP$THE$LINKS PERFORMS THE FOILOWING FUNCTIONS:            *
                *                                                               *
                *   1.  SNAPS THE OUTGOING AND INCOMING LINKS FOR A PROCEDURE   *
                *       OBJECT.                                                 *
                *                                                               *
                *   2.  SNAPS THE OUTGOING LINK FOR A DATA OBJECT.              *
                *                                                               *
                *****************************************************************/
```

```
 128   1         SNAP$THE$LINKS : PROCEDURE;

 129   2            IF OBJECT$TYPE = TYPE$PROCEDURE THEN DO;

                       /* SNAP A LINK FOR AN EXTERNAL PROCEDURE */

 131   3               OUTGOING$LINK (0) = JUMP$TO;
 132   3               OUTGOING$LINK (1) = LOW (IN$LINK$ADDRESS);
 133   3               OUTGOING$LINK (2) = HIGH (IN$LINK$ADDRESS);

 134   3               IF INCOMING$LINK.LOAD$LP (0) = UNSNAPPED THEN DO;
 136   4                 INCOMING$LINK.LOAD$LP (0) = LOAD$LP$INST;
 137   4                 INCOMING$LINK.LOAD$LP (1) = LOW (NEW$LINK$PTR);
 138   4                 INCOMING$LINK.LOAD$LP (2) = HIGH (NEW$LINK$PTR);

 139   4                 INCOMING$LINK.JUMP$INST (0) = JUMP$TO;
 140   4                 INCOMING$LINK.JUMP$INST (1) = LOW (TARGET$ADDRESS);
 141   4                 INCOMING$LINK.JUMP$INST (2) = HIGH (TARGET$ADDRESS);
 142   4               END;        /* OF THE IF INCOMING$LINK IS UNSNAPPED CLAUSE */

 143   3            END;   /* OF THE THEN  CLAUSE */

 144   2            ELSE DO;

                       /* SNAP A DATA LINK */

 145   3               OUTGOING$LINK (0) = LOAD$POINTER;
 146   3               OUTGOING$LINK (1) = LOW (TARGET$ADDRESS);
 147   3               OUTGOING$LINK (2) = HIGH (TARGET$ADDRESS);
 148   3               OUTGOING$LINK (3) = RETURN$INST;

 149   3            END;   /* OF THE ELSE CLAUSE */

 150   2         END SNAP$THE$LINKS;
```

```
                /*********************************************************************
                 *                                                                   *
                 *    LINKER IS THE CONTROL MODULE CALLED TO PERFORM THE             *
                 *    LINKING PROCESS.                                               *
                 *                                                                   *
                 *********************************************************************/

 151   1        LINKER : PROCEDURE (LINK$PTR, SYM$NAME$OFFSET) ADDRESS;

 152   2            DECLARE LINK$PTR POINTER,
                            SYM$NAME$OFFSET INTEGER;

                    /*** FIRST INITIALIZE THE LINKAGE POINTER AND SYMBOLIC NAME
                         OFFSET.  ***/

 153   2            LINKAGE$POINTER = LINK$PTR;
 154   2            SN$OFFSET = SYM$NAME$OFFSET;

 155   2            CALL ACCESS$SYMBOLIC$NAME$DATA (LINKAGE$POINTER, SN$OFFSET);

 156   2            CALL ASM$MAKE$ACCESSABLE (SN$BUFFER$POINTER, OBJECT$ID$POINTER);

 157   2            CALL LINKAGE$TABLE$ROUTINES (OBJECT.UNIQUE$ID, OBJECT.BASE$ADDRESS,
                                                  OBJECT$TYPE, SN$BUFFER$POINTER);

 158   2            CALL ACCESS$ENTRY$NAME$DATA;

 159   2            CALL SNAP$THE$LINKS;


 160   2            RETURN OUT$LINK$ADDRESS;

 161   2        END LINKER;
```

```
                  /**********************************************************
                   *                                                        *
                   *   INITIALIZE$LINKER PERFORMS THE FOLLOWING FUNCTIONS:  *
                   *                                                        *
                   *   1.  INITIALIZES THE LINKER.                          *
                   *                                                        *
                   **********************************************************/

162   1           INITIALIZE$LINKER : PROCEDURE (RET$VAL$PTR) PUBLIC;

163   2             DECLARE RET$VAL$PTR POINTER,
                            RET$VALUE BASED RET$VAL$PTR STRUCTURE (
                               LINKER$ADDRESS ADDRESS,
                               LINK$ADDR$TABLE$BASE ADDRESS,
                               LINK$TABLE$ADDRESS ADDRESS),
                            I BYTE;

164   2             OBJECT$ID$POINTER = .OBJECT.UNIQUE$ID;

165   2             DO I = 0 TO 15;
166   3               LINKAGE$ADDRESS$TABLE (I).VALID$BIT = INVALID;
167   3             END;

168   2             FREE$LINK$TABLE = .LINKAGE$TABLE (0);
169   2             SN$BUFFER$POINTER = .SN$BUFFER.NAME (0);
170   2             EN$BUFFER$PTR = .EN$BUFFER.NAME (0);

                    /*** WE RETURN TO PROCESS INITIALIZATION THE ADDRESS OF
                         THE SUBROUTINE "LINKER", THE ADDRESS OF THE
                         LINKAGE ADDRESS TABLE AND THE LINKAGE TABLE. ***/

171   2             RET$VALUE.LINKER$ADDRESS = .LINKER;
172   2             RET$VALUE.LINK$ADDR$TABLE$BASE =
                                      .LINKAGE$ADDRESS$TABLE (0).VALID$BIT;
173   2             RET$VALUE.LINK$TABLE$ADDRESS = .LINKAGE$TABLE (0);
```

```
 174   2          END INITIALIZE$LINKER;


 175   1       END DLKR;



MODULE INFORMATION:

     CODE AREA SIZE     = 0519H   1305D
     VARIABLE AREA SIZE = 0492H   1170D
     MAXIMUM STACK SIZE = 0006H      6D
     595 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

```
PL/M-80 COMPILER    ADDRESS SPACE MANAGER

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE ASM
OBJECT MODULE PLACED IN :F1:ASM.OBJ
COMPILER INVOKED BY:  PLM80 :F1:ASM.SRC PAGELENGTH(38) TITLE('ADDRESS SPACE MANAGER')


   1          ASM : DO;

              /* DATE LAST EDITED : 4 AUGUST 1980 */

   2   1      DECLARE LIT LITERALLY 'LITERALLY',
                      TRUE LIT '01H',
                      FALSE LIT '00H',
                      SPACE LIT '20H',
                      FORM$FEED LIT '0CH',
                      VALID LIT '01H',
                      INVALID LIT '00H',
                      POINTER LIT 'ADDRESS',
                      INTEGER LIT 'ADDRESS',
                      FUNCTION LIT 'PROCEDURE',
                      BOOLEAN LIT 'BYTE';


   3   1      DECLARE

                 PRT$SIZE INTEGER,
                 PRT (16) STRUCTURE (
                   VALID$BIT BOOLEAN,
                   NAME (12) BYTE,
                   BASE$ADDR ADDRESS),
                 FREE$MEMORY ADDRESS;

              /*****************************************************************
               *                                                                *
               *        EXTERNALLY DEFINED SYSTEM PROCEDURE DECLARATIONS        *
```

```
                *                                                                  *
                *******************************************************************/

                /*** OPEN$FILE OPENS A FIIE ON DISK.  ***/

  4   1         OPEN$FILE : PROCEDURE (PTR$TO$FILENAME) EXTERNAL;
  5   2           DECLARE PTR$TO$FILENAME POINTER;
  6   2         END OPEN$FILE;

                /*** CLOSE$FILE CLOSES A FILE ON DISK.  ***/

  7   1         CLOSE$FILE : PROCEDURE EXTERNAL;
  8   2         END CLOSE$FILE;

                /*** READ$DISK READS 128 BYTES FROM A FIIE ON DISK INTO A BUFFER
                   STARTING AT LOCATION BUFFER$ADDR. ***/

  9   1         READ$DISK : FUNCTION (BUFFER$ADDR) BOOLEAN EXTERNAL;
 10   2           DECLARE BUFFER$ADDR ADDRESS;
 11   2         END READ$DISK;

                /*** DISPLAY$CHAR OUTPUTS AN ASCII CHARACTER TO THE CRT. ***/

 12   1         DISPLAY$CHAR : PROCEDURE (CHARACTER) EXTERNAL;
 13   2           DECLARE CHARACTER BYTE;
 14   2         END DISPLAY$CHAR;

                /*** DISPLAY OUTPUTS AN ASCII CHARACTER STRINT TO THE CRT. ***/

 15   1         DISPLAY : PROCEDURE (STRING$ADDRESS) EXTERNAL;
 16   2            DECLARE STRING$ADDRESS ADDRESS;
 17   2         END DISPLAY;

                /*** OUTPUT$ADDR DISPLAYS A 2-BYTE VALUE ON THE CRT. ***/
```

```
 18   1      OUTPUT$ADDR : PROCEDURE (DEVICE, VALUE) EXTERNAL;
 19   2         DECLARE VALUE ADDRESS,
                       DEVICE BYTE;
 20   2      END OUTPUT$ADDR;

             /*** CRLF GENERATES A CARRIAGE RETURN AND LINE FEED ON THE CRT. ***/

 21   1      CRLF : PROCEDURE EXTERNAL;
 22   2      END CRLF;

             /*** END OF EXTERNAL SYSTEM DECLARATIONS. ***/

             /*****************************************************************/


             /*    T H E   A D D R E S S   S P A C E   M A N A G E R    */



                  /*** LOAD$OBJECT AND RELOCATE ARE INTERFACE ROUTINES
                  BETWEEN THE ADDRESS SPACE MANAGER AND THE CP/M OPERATING
                  SYSTEM.   ***/

             /*****************************************************************
              *                                                               *
              *   RELOCATE PERFORMS THE FOLLOWING FUNCTIONS:                  *
              *   1.  CHANGES ALL RELATIVE ADDRESSES IN A PROCEDURE TO        *
              *         ABSOLUTE ADDRESSES.                                   *
              *                                                               *
              *****************************************************************/


 23   1      RELOCATE : PROCEDURE (OBJ$NAME$PTR, BASE$ADDRESS);
```

```
 24   2          DECLARE OBJ$NAME$PTR POINTER,
                         OBJECT$NAME BASED OBJ$NAME$PTR (12) BYTE,
                         TEMP$NAME$BUFFER (12) BYTE,
                         TEMP$NAME$PTR POINTER,
                         BASE$ADDRESS ADDRESS,
                         FILE$POINTER POINTER,
                         RELATIVE$ADDR BASED FILE$POINTER ADDRESS,
                         RELOC$BUFF$PTR POINTER,
                         RELOC$BUFFER (128) BYTE,
                         ADDRESS$VALUE BASED RELOC$BUFF$PTR ADDRESS,
                         NUM$OF$RELOC$BYTES INTEGER,
                         I BYTE;


                 /* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . */

                 /*** LOAD$RELOC$BUFFER LOADS 128 BYTES OF RELOCATION
                      BITS INTO THE RELOCATION BUFFER.  ***/

 25   2               LOAD$RELOC$BUFFER : PROCEDURE;
 26   3                 DECLARE DUMMY BYTE;
 27   3                 DUMMY = READ$DISK (RELOC$BUFF$PTR);
 28   3               END LOAD$RELOC$BUFFER;


                 /* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . */

                 /*** RELOC$8$BYTES RELOCATES EIGHT BYTES IN THE EXECUTABLE
                      OBJECT FILE.  ***/

 29   2               RELOC$8$BYTES : PROCEDURE (SUBSCRIPT);
 30   3                 DECLARE SUBSCRIPT BYTE,
                                BYTE$MASK BYTE,
                                LOOP BYTE;
```

```
  31   3              BYTE$MASK = 80H;
  32   3              DO LOOP = 1 TO 8;

                       /* IF THE RELOCATION BIT IS 1, THEN RELOCATE */

  33   4                IF (RELOC$BUFFER (SUBSCRIPT) AND BYTE$MASK) <> 0 THEN
  34   4                  RELATIVE$ADDR = RELATIVE$ADDR + BASE$ADDRESS - 100H;

                        /* NOW SHIFT THE BYTE$MASK BIT TO THE RIGHT AND
                           INCREMENT THE FILE$POINTER. */

  35   4                BYTE$MASK = SHR (BYTE$MASK, 1);
  36   4                FILE$POINTER = FILE$POINTER + 1;

  37   4              END;   /* OF THE LOOP */

  38   3            END RELOC$8$BYTES;

         /* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . */

         /*** BEGIN RELOCATION ***/

         /*** SET FILE$POINTER = THE BASE ADDRESS OF THE OBJECT
              FILE AND RELOC$BUFF$PTR TO POINT TO THE RELOCATION
              BUFFER.  ALSO SET TEMP$NAME$PTR TO POINT TO THE
              TEMP$NAME$BUFFER.  THE TEMPORARY NAME BUFFER WILL
              CONTAIN THE OBJECT NAME AND IS USED TO PREVENT
              SETTING THE OBJECT TYPE TO 'RLB' BY SETTING THE
              TYPE IN THE TEMPORARY BUFFER TO 'RLB' (RELOCATION
              BITS).  ***/

  39   2     FILE$POINTER = BASE$ADDRESS;
  40   2     RELOC$BUFF$PTR = .RELOC$BUFFER(0);
```

```
 41   2          TEMP$NAME$PTR = .TEMP$NAME$BUFFER(0);

                 /* NOW SET TEMP$NAME$BUFFER TO OBJECT$NAME */

 42   2          DO I = 0 TO 11;
 43   3            TEMP$NAME$BUFFER (I) = OBJECT$NAME(I);
 44   3          END;


                 /*** SET UP AND OPEN THE RELOCATION BITS FILE ***/

 45   2          I = 0;
 46   2          DO WHILE TEMP$NAME$BUFFER(I) <> '.';
 47   3            I = I + 1;
 48   3          END;

                 /* SET FILE TYPE TO 'RLB' */

 49   2          TEMP$NAME$BUFFER(I := I + 1) = 'R';
 50   2          TEMP$NAME$BUFFER(I := I + 1) = 'L';
 51   2          TEMP$NAME$BUFFER(I := I + 1) = 'B';

 52   2          CALL OPEN$FILE (TEMP$NAME$PTR);

                 /***  START THE RELOCATION ***/

 53   2          I = 2;      /* INITIALIZE THE SUBSCRIPT TO 2 BECAUSE THE FIRST
                               TWO BYTES OF THE RELOCATION BITS FILE CONTAIN
                               THE NUMBER OF RELOCATION BYTES IN THE FILE. */

 54   2          CALL LOAD$RELOC$BUFFER;

                 /***  EXTRACT THE SIZE OF THE RELOCATION BITS FILE ***/
```

```
  55   2          NUM$OF$RELOC$BYTES = ADDRESS$VALUE;

                  /*** RELOCATE 8 BYTES IN THE OBJECT FILE ***/

  56   2          DO WHILE I <= NUM$OF$RELOC$BYTES;

  57   3            CALL RELOC$8$BYTES (I);
  58   3            I = I + 1;

  59   3            IF I = 128 THEN DO;

                      /* IF THE NUM$OF$RELOC$BYTES IS 127, THEN RELOCATION
                         IS COMPLETE AND NO FURTHER COMPUTATIONS IS NECESSARY. */

  61   4              IF NUM$OF$RELOC$BYTES <> 127 THEN
  62   4                DO;
  63   5                  NUM$OF$RELOC$BYTES = NUM$OF$RELOC$BYTES - 128;
  64   5                  CALL LOAD$RELOC$BUFFER;
  65   5                  I = 0;
  66   5                END;   /* OF THE IF NUM$OF$RELOC$BYTES <> 127 CLAUSE */

  67   4            END;   /* OF THE IF I = 128 CLAUSE */

  68   3          END;      /* OF THE WHILE CLAUSE */

                  /***  THE RELOCATION IS COMPLETE--CLOSE THE RELOCATION
                        BITS FILE.    ***/

  69   2          CALL CLOSE$FILE;

  70   2        END RELOCATE;

                /**********************************************************
                 *                                                        *
                 * LOAD$OBJECT PERFORMS THE FOLLOWING FUNCTIONS:          *
```

```
                  *                                                            *
                  * 1. LOADS A FILE WHOSE NAME IS POINTED TO BY OBJECT NAME    *
                  *    POINTER (OBJ$NAME$PTR) INTO MEMORY  AT THE NEXT FREE MEMORY *
                  *    LOCATION.                                               *
                  *                                                            *
                  * 2. UPDATES THE NEXT FREE MEMORY LOCATION (FREE$MEMORY)     *
                  *                                                            *
                  * 3. IF THE FILE IS AN EXECUTABLE FILE (I.E., FILE TYPE = COM), *
                  *    CALL PROCEDURE RELOCATE AND RELOCATES THE FILE.         *
                  *                                                            *
                  ***************************************************************/

 71   1          LOAD$OBJECT : FUNCTION (OBJ$NAME$PTR) ADDRESS;

 72   2            DECLARE  OBJ$NAME$PTR POINTER,
                            OBJECT$NAME BASED OBJ$NAME$PTR (12) BYTE,
                            BASE$ADDRESS ADDRESS,
                            I BYTE;

                   /*** OPEN THE OBJECT FILE ***/

 73   2            CALL OPEN$FILE (OBJ$NAME$PTR);

                   /*** SET BASE$ADDRESS = THE BASE LOCATION OF THE OBJECT
                        AND LOAD THE OBJECT INTO MEMORY.  ***/

 74   2            BASE$ADDRESS = FREE$MEMORY;

 75   2            DO WHILE READ$DISK (FREE$MEMORY) = TRUE;

                     /***  INCREMENT FREE$MEMORY AND LOAD ANOTHER 128 BYTES ***/

 76   3              FREE$MEMORY = FREE$MEMORY + 128;

 77   3            END;   /* OF THE WHILE CLAUSE */
```

```
                 /*** NOW CLOSE THE OBJECT FILE ***/

  78   2         CALL CLOSE$FILE;

                 /*** IF THE OBJECT WAS EXECUTABLE CODE, THEN PERFORM A RELOCATION ***/

  79   2         I = 0;
  80   2         DO WHILE OBJECT$NAME(I) <> '.';
  81   3           I = I + 1;
  82   3         END;

  83   2         IF OBJECT$NAME(I := I + 1) = 'C'  AND
                    OBJECT$NAME(I := I + 1) = 'O'  AND
                    OBJECT$NAME(I := I + 1) = 'M'

                      THEN CALL REIOCATE (OBJ$NAME$PTR, BASE$ADDRESS);

                 /*** RETURN-LIST : BASE$ADDRESS, FREE MEMORY ***/

  85   2         RETURN BASE$ADDRESS;

  86   2      END LOAD$OBJECT;

              /****************************************************************
               *                                                              *
               * COMPARE PERFORMS THE FOLLOWING FUNCTIONS:                    *
               *                                                              *
               * 1.  DETERMINES IF THE OBJECT NAME PASSED AS AN ACTUAL        *
               *     PARAMETER IS EQUAL TO PRT(PRT$INDEX).NAME.               *
               *                                                              *
               ****************************************************************/

  87   1      COMPARE : FUNCTION (OBJ$NAME$PTR, PRT$INDEX) BOOLEAN;
```

```
  88   2          DECLARE OBJ$NAME$PTR POINTER,
                          OBJ$NAME BASED OBJ$NAME$PTR (12) BYTE,
                          PRT$INDEX BYTE,
                          CHECK$RESULT BOOLEAN,
                          (J,I) BYTE;

  89   2          J = 0;
  90   2          CHECK$RESULT = TRUE;

                  /*** PERFORM A BYTE BY BYTE COMPARISON OF OBJ$NAME AND
                       PRT(PRT$INDEX).NAME TO DETERMINE WHETHER THEY MATCH.
                       DO NOT LOOK PAST THE FILE TYPE FOR THE COMPARISON. ***/

  91   2          DO WHILE CHECK$RESULT AND OBJ$NAME(J) <> '.';

  92   3            IF OBJ$NAME(J) <> PRT(PRT$INDEX).NAME(J)
                      THEN CHECK$RESULT = FALSE;

  94   3            J = J + 1;

  95   3          END;   /* OF THE WHILE CLAUSE */

                  /*** IF THE OBJECT$NAME WAS A MATCH, THEN CHECK FOR A
                       MATCH OF THE OBJECT TYPE.  ***/

  96   2          IF CHECK$RESULT THEN

  97   2             DO I = (J + 1) TO (J + 3);
  98   3                IF OBJ$NAME(I) <> PRT(PRT$INDEX).NAME(I) THEN
  99   3                   CHECK$RESULT = FALSE;
 100   3             END;

 101   2          RETURN CHECK$RESULT;

 102   2       END COMPARE;
```

```
                /*******************************************************************
                 *                                                                 *
                 *   ASM$MAKE$ACCESSABLE PERFORMS THE FOLLOWING FUNCTIONS:         *
                 *                                                                 *
                 *   1.  DETERMINES IF THE OBJECT IS ALREADY IN THE PROCESS        *
                 *       REFERENCE TABLE (I.E., THE OBJECT HAS ALREADY BEEN        *
                 *       MADE ACCESSABLE).                                         *
                 *                                                                 *
                 *   2.  IF NOT, LOADS THE OBJECT INTO MEMORY AND ENTERS IT IN     *
                 *       THE PROCESS REFERENCE TABLE.                              *
                 *                                                                 *
                 *   3.  RETURNS TO THE POINT OF CALL A POINTER TO THE UNIQUE      *
                 *       ID AND BASE ADDRESS OF THE OBJECT.                        *
                 *                                                                 *
                 *******************************************************************/

103   1         ASM$MAKE$ACCESSABLE : PROCEDURE (OBJ$NAME$PTR, RETURN$VALUE$PTR)
                                          PUBLIC;

104   2           DECLARE  OBJ$NAME$PTR POINTER,
                           OBJECT$NAME BASED OBJ$NAME$PTR (12) BYTE,
                           RETURN$VALUE$PTR POINTER,
                           RETURN$VALUE BASED RETURN$VALUE$PTR STRUCTURE (
                             UNIQUE$ID BYTE,
                             BASE$ADDR ADDRESS),
                           FOUND BOOLEAN,
                           OBJECT$SUBSCRIPT BYTE,
                           I BYTE,
                           J BYTE;


105   2           I = 0;
106   2           FOUND = FALSE;
```

```
                     /*** CHECK TO SEE IF OBJECT$NAME IS IN THE PRT. ***/

 107   2             DO WHILE NOT FOUND AND I < PRT$SIZE;

 108   3               IF PRT(I).VALID$BIT = VALID THEN
 109   3                   IF COMPARE ( OBJ$NAME$PTR,I) THEN DO;
 111   4                      FOUND = TRUE;
 112   4                      OBJECT$SUBSCRIPT = I;
 113   4                   END;

 114   3               I = I +  1;
 115   3             END;   /* OF THE WHILE CLAUSE */

 116   2             IF NOT FOUND THEN
 117   2               DO;
 118   3                 I = 0;

                         /* FIND A FREE PRT ENTRY */

 119   3                 DO WHILE PRT(I).VALID$BIT;
 120   4                   I = I + 1;
 121   4                 END;

 122   3                 OBJECT$SUBSCRIPT = I;

                         /*  LOAD THE OBJECT INTO THE ADDRESS SPACE AND
                             SET UP A PRT ENTRY FOR THE OBJECT.  */

 123   3                 PRT(OBJECT$SUBSCRIPT).VALID$BIT = VALID;

 124   3                 DO J = 0 TO 11;
 125   4                   PRT(OBJECT$SUBSCRIPT).NAME(J) = OBJECT$NAME(J);
 126   4                 END;
```

```
 127   3              PRT(OBJECT$SUBSCRIPT).BASE$ADDR = LOAD$OBJECT(OBJ$NAME$PTR);

 128   3           END;   /* OF THE IF NOT FOUND CLAUSE */

                   /*** NOW SET UP THE RETURN VALUE STRUCTURE ***/

 129   2           RETURN$VALUE.UNIQUE$ID = OBJECT$SUBSCRIPT;
 130   2           RETURN$VALUE.BASE$ADDR = PRT(OBJECT$SUBSCRIPT).BASE$ADDR;

 131   2       END ASM$MAKE$ACCESSABLE;

               /****************************************************************
                *                                                              *
                *  ASM$REMOVE$SEG PERFORMS THE FOLLOWING FUNCTIONS:            *
                *                                                              *
                *  1. REMOVES AN OBJECT FROM A PROCESS ADDRESS SPACE BY        *
                *     DELETING IT FROM THE PRT.                                *
                *                                                              *
               ****************************************************************/

 132   1       ASM$REMOVE$SEG : PROCEDURE (OBJ$NAME$PTR) PUBLIC;

 133   2          DECLARE OBJ$NAME$PTR POINTER,
                          OBJECT$NAME BASED OBJ$NAME$PTR (12) BYTE,
                          FOUND BOOLEAN,
                          OBJECT$SUBSCRIPT BYTE,
                          J BYTE,
                          I BYTE;

 134   2          I = 0;
 135   2          FOUND = FALSE;

                  /*** FIND THE OBJECT IN THE PRT ***/
```

```
136   2         DO WHILE NOT FOUND AND I < PRT$SIZE;

137   3           IF PRT(I).VALID$BIT = VALID THEN
138   3               IF COMPARE(OBJ$NAME$PTR, I) THEN DO;
140   4                  FOUND = TRUE;
141   4                  OBJECT$SUBSCRIPT = I;
142   4               END;

143   3           I = I + 1;
144   3         END;   /* OF THE WHILE CLAUSE */

                /*** REMOVE THE OBJECT ***/

145   2         PRT(OBJECT$SUBSCRIPT).VALID$BIT = INVALID;

146   2      END ASM$REMOVE$SEG;

             /**************************************************************
              *                                                            *
              *  INITIALIZE$ASM PERFORMS THE FOLLOWING FUNCTIONS:          *
              *                                                            *
              *  1.  INITIALIZES THE ADDRESS SPACE MANAGER DURING PROCESS  *
              *      INITIALIZATION.                                       *
              *                                                            *
              **************************************************************/

147   1      INITIALIZE$ASM : PROCEDURE PUBLIC;

148   2         DECIARE I BYTE;

149   2         DO I = 0 TO 15;
150   3           PRT(I).VALID$BIT = INVALID;
151   3         END;

152   2         FREE$MEMORY = .MEMORY;
```

```
 153   2         PRT$SIZE = 16;

 154   2       END INITIALIZE$ASM;

               /***     E N D   O F   A D D R E S S   S P A C E   M A N A G E R ***/

               /*************************************************************/

               /*** THE FOLLOWING PROCEDURE DISPLAYS THE PROCESS REFERENCE TABLE AND
                    IS NOT NECESSARY FOR THE PROPER EXECUTION OF THE ADDRESS SPACE
                    MANAGER OR THE DYNAMIC LINKER--IT IS STRICTLY FOR THE PURPOSE
                    OF THE DEMONSTRATION.   ***/

               /*************************************************************/


                /************************************************************
                 *                                                          *
                 *   DISPLAY$PRT PERFORMS THE FOLLOWING FUNCTIONS:          *
                 *                                                          *
                 *   1.  DISPLAYS THE PROCESS REFERENCE TABLE ON THE CRT.   *
                 *                                                          *
                 ************************************************************/

 155   1        DISPLAY$PRT : PROCEDURE PUBLIC;

 156   2         DECLARE (I,J,K) BYTE;

                 /*** OUTPUT THE HEADING "PROCESS REFERENCE TABLE". ***/

 157   2         CALL DISPLAY$CHAR (FORM$FEED);
```

```
 158   2          CALL CRLF;
 159   2          CALL CRLF;
 160   2          CALL CRLF;
 161   2          CALL DISPLAY(.('     THE PROCESS REFERENCE TABLE','$'));
 162   2          CALL CRLF;
 163   2          CALL DISPLAY(.('     ---------------------------','$'));
 164   2          CALL CRLF;
 165   2          CALL CRLF;

                  /*** STEP THROUGH THE PROCESS REFERENCE TABLE AN ENTRY AT A
                       TIME.  IF THE VALID$BIT IS VALID, THEN DISPLAY THE
                       ENTRY.  ELSE DISPLAY 'NO ENTRY'.     ***/

 166   2          DO I = 1 TO PRT$SIZE;

                     /* FIRST DISPLAY THE PRT SUBSCRIPT (I) */

 167   3             CALL DISPLAY(.('     ','$'));
 168   3             IF I < 10 THEN CALL DISPLAY$CHAR(SPACE);
 170   3             CALL OUTPUT$ADDR(0, DOUBLE(I));
 171   3             CALL DISPLAY(.(' : ','$'));

                     /* NOW DISPLAY THE PRT ENTRY ITSELF */

 172   3             IF PRT(I - 1).VALID$BIT = INVALID THEN
 173   3               CALL DISPLAY(.('NO ENTRY','$'));
 174   3             ELSE DO;
 175   4                 CALL DISPLAY(.('OBJECT NAME  - ','$'));

                         /* DISPLAY THE OBJECT NAME */

 176   4                 J = 0;
 177   4                 DO WHILE PRT(I - 1).NAME(J) <> '.';

                           /* DISPLAY THE FILE NAME */
```

```
 178   5                  CALL DISPLAY$CHAR(PRT(I - 1).NAME(J));
 179   5                  J = J + 1;
 180   5                END;

 181   4                DO K = J TO (J + 3);

                            /* DISPLAY THE FILE TYPE */

 182   5                    CALL DISPLAY$CHAR(PRT(I - 1).NAME(K));
 183   5                END;

 184   4              CALL CRLF;
 185   4              CALL DISPLAY(.('          BASE ADDRESS - ','$'));
 186   4              CALL OUTPUT$ADDR(0, PRT(I - 1).BASE$ADDR);
 187   4            END;   /*  OF THE ELSE CLAUSE */

 188   3            CALL CRLF;

 189   3         END;   /* OF THE DO I = 0 TO PRT$SIZE IOOP */

 190   2        END DISPLAY$PRT;


 191   1       END ASM;


MODULE INFORMATION:

     CODE AREA SIZE     = 0628H   1576D
     VARIABLE AREA SIZE = 01AEH    430D
     MAXIMUM STACK SIZE = 000AH     10D
     579 LINES READ
     0 PROGRAM ERROR(S)
```

```
END OF PL/M-80 COMPILATION
```

```
ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE DISLT
OBJECT MODULE PLACED IN :F1:DISLT.OBJ
COMPILER INVOKED BY:  PLM80 :F1:DISLT.SRC PAGELENGTH(38) TITLE('DISPLAY LINKAGE TABLE')



   1          DISLT : DO;

              /* DATE LAST EDITED : 4 AUGUST 1980 */

              /* THIS ROUTINE DISPLAYS THE LINKAGE ADDRESS TABLE AND LINKAGE
                 TABLE ON THE CRT. */

   2   1      DECLARE LIT LITERALLY 'LITERALLY',
                      POINTER LIT 'ADDRESS',
                      INTEGER LIT 'ADDRESS',
                      BOOLEAN LIT 'BYTE',
                      TRUE LIT '01H',
                      FALSE LIT '00H',
                      SPACE LIT '20H',
                      FORM$FEED LIT '0CH',
                      LITTLE$P LIT '70H',
                      BAR LIT '7CH',

                      PUSH$D LIT '0D5H',
                      LOAD$IP LIT '01H',
                      LOAD$PTR LIT '11H',
                      JUMP$TO LIT '0C3H',

                      INCOMING LIT '00H',
                      OUTGOING LIT '01H',
                      VALID LIT '01H',
                      INVALID LIT '00H';

   3   1      DECLARE COUNTER BYTE INITIAL (01H);
```

```
              /********************************************************************
               *                                                                  *
               *        EXTERNALLY DEFINED SYSTEM PROCEDURE DECLARATIONS          *
               *                                                                  *
               ********************************************************************/

               /*** DISPLAY OUTPUTS AN ASCII CHARACTER STRING TO THE CRT.  ***/

   4   1       DISPLAY : PROCEDURE (STRING$ADDRESS) EXTERNAL;
   5   2         DECLARE STRING$ADDRESS POINTER;
   6   2       END DISPLAY;

               /*** OUTPUT$ADDR DISPLAYS A 2-BYTE VALUE ON THE CRT. ***/

   7   1       OUTPUT$ADDR : PROCEDURE (DEVICE, VALUE) EXTERNAL;
   8   2         DECLARE DEVICE BYTE,
                         VALUE ADDRESS;
   9   2       END OUTPUT$ADDR;

               /*** DISPLAY$CHAR OUTPUTS AN ASCII CHARACTER TO THE CRT ***/

  1Ø   1       DISPLAY$CHAR : PROCEDURE (CHARACTER) EXTERNAL;
  11   2         DECLARE CHARACTER BYTE;
  12   2       END DISPLAY$CHAR;

               /*** CRLF GENERATES A CARRIAGE RETURN AND LINE FEED ON THE CRT. ***/

  13   1       CRLF : PROCEDURE EXTERNAL;
  14   2       END CRLF;

               /***     END OF EXTERNAL SYSTEM DECLARATIONS    ***/

               /*********************************************************************/
```

```
                /***                         USER ROUTINES                         ***/

                /*********************************************************************/

                /*** DISPLAY$HEX OUTPUTS A BYTE VALUE IN HEXIDECIMAL FORM TO THE CRT ***/

  15    1       DISPLAY$HEX : PROCEDURE (VALUE);

  16    2          DECLARE VALUE BYTE,
                           TEMP$VAL BYTE;

  17    2          TEMP$VAL = SHL((VALUE AND 0F0H), 4);

  18    2          IF TEMP$VAL < 10 THEN CALL DISPLAY$CHAR(TEMP$VAL + 30H);
  20    2          ELSE CALL DISPLAY$CHAR(TEMP$VAL + 37H);

  21    2          VALUE = VALUE AND 0FH;

  22    2          IF VALUE < 10 THEN CALL DISPLAY$CHAR(VALUE + 30H);
  24    2          ELSE CALL DISPLAY$CHAR(VALUE + 37H);

  25    2          CALL DISPLAY$CHAR ('H');
  26    2       END DISPLAY$HEX;

                /***  LINE$OF$DOTS AND LINE$OF$DASHES DISPLAYS A LINE OF DOTS OR
                      DASHES ON THE CRT. ***/

  27    1       LINE$OF$DOTS : PROCEDURE;

  28    2         CALL CRLF;
  29    2         CALL DISPLAY (.('     ', BAR, '................', BAR, '$'));
  30    2         CALL CRLF;

  31    2       END LINE$OF$DOTS;
```

```
 32   1        LINE$OF$DASHES : PROCEDURE;

 33   2          CALL CRIF;
 34   2          CALL DISPLAY (.('     ', BAR, '----------------', BAR, '$'));
 35   2          CALL CRIF;

 36   2        END LINE$OF$DASHES;

               /*** PRINT$ADDRESS DISPLAYS ' ADDRESS ' FOLLOWED BY VALUE ***/

 37   1        PRINT$ADDRESS : PROCEDURE (VALUE);

 38   2           DECLARE VALUE ADDRESS;

 39   2           CALL DISPLAY (.(' (ADDRESS - ','$'));
 40   2           CALL OUTPUT$ADDR (0, VALUE);
 41   2           CALL DISPLAY$CHAR (')');

 42   2        END PRINT$ADDRESS;

               /*** PRINT$VALUE PRINTS AN INTEGER ON THE CRT AND FILLS IN THE
                    NUMBER OF NECESSARY SPACES TO KEEP THE OUTPUT UNIFORM. ***/

 43   1         PRINT$VALUE : PROCEDURE (DEVICE, NUMBER);

 44   2            DECLARE NUMBER ADDRESS,
                            DEVICE BYTE;

 45   2            CALL OUTPUT$ADDR (DEVICE, NUMBER);

 46   2            IF NUMBER < 10 THEN CALL DISPLAY(.('    ','$'));
                    ELSE
 48   2            IF NUMBER < 100 THEN CALL DISPLAY (.(SPACE,SPACE,SPACE,'$'));
 50   2            ELSE IF NUMBER < 1000 THEN CALL DISPLAY (.(SPACE, SPACE, '$'));
 52   2            ELSE IF NUMBER < 10000 THEN CALL DISPLAY$CHAR (SPACE);
```

```
                    END PRINT$VALUE;

                   /*** DISPLAY$PROC$LINK OUTPUTS A SNAPPED PROCEDURE OUTGOING LINK
                        TO THE CRT. ***/

  55   1           DISPLAY$PROC$LINK : PROCEDURE (OUT$LINK$ADDR);

  56   2             DECIARE OUT$LINK$ADDR POINTER,
                             SNAPPED$LINK BASED OUT$LINK$ADDR STRUCTURE (
                                JUMP$INST BYTE,
                                IN$LINK$ADDR ADDRESS,
                                FILLER ADDRESS);

  57   2             CALI DISPLAY (.('     ', BAR, '  JUMP TO ','$'));
  58   2             CALI PRINT$VALUE (0, SNAPPED$LINK.IN$LINK$ADDR);
  59   2             CALL DISPLAY (.(SPACE, BAR, '  SNAPPED PROCEDURE LINK','$'));

  60   2             CALL PRINT$ADDRESS (OUT$LINK$ADDR);
  61   2             CALI LINE$OF$DASHES;

  62   2          END DISPLAY$PROC$LINK;

                  /*** DISPLAY$DATA$LINK OUTPUTS A SNAPPED OUTGOING DATA LINK TO THE
                       CRT. ***/

  63   1          DISPLAY$DATA$LINK : PROCEDURE (OUT$LINK$ADDR);

  64   2             DECIARE OUT$LINK$ADDR POINTER,
                             SNAPPED$LINK BASED OUT$LINK$ADDR STRUCTURE (
                                LOAD$PTR$INST BYTE,
                                DATA$ADDRESS ADDRESS,
                                RETURN$INST BYTE);
```

```
 65   2           CALL DISPLAY (.('     ', BAR, ' LOAD PTR ','$'));
 66   2           CALL PRINT$VALUE (0, SNAPPED$LINK.DATA$ADDRESS);
 67   2           CALL DISPLAY (.(SPACE, BAR, '  SNAPPED DATA LINK','$'));

 68   2           CALL PRINT$ADDRESS (OUT$LINK$ADDR);
 69   2           CALL LINE$OF$DOTS;
 70   2           CALL DISPLAY (.('     ', BAR, '     RETURN     ', BAR, '$'));
 71   2           CALL LINE$OF$DASHES;

 72   2        END DISPLAY$DATA$LINK;

               /*** DISPLAY$INCOMING$LINK OUTPUTS A SNAPPED INCOMING LINK TO THE
                    CRT. ***/

 73   1        DISPLAY$INCOMING$LINK : PROCEDURE (IN$LINK$ADDR);

 74   2           DECLARE IN$LINK$ADDR POINTER,
                          SNAPPED$LINK BASED IN$LINK$ADDR STRUCTURE (
                             LOAD$LP$INST BYTE,
                             LINK$PTR ADDRESS,
                             JUMP$INST BYTE,
                             TARGET$ADDR ADDRESS);

 75   2           CALL DISPLAY(.('     ', BAR, '  LOAD LP ','$'));
 76   2           CALL PRINT$VALUE (0, SNAPPED$LINK.LINK$PTR);
 77   2           CALL DISPLAY(.(SPACE, BAR, '  INCOMING LINK','$'));
 78   2           CALL PRINT$ADDRESS (IN$LINK$ADDR);
 79   2           CALL LINE$OF$DOTS;

 80   2           CALL DISPLAY(.('     ', BAR, '  JUMP TO ','$'));
 81   2           CALL PRINT$VALUE (0, SNAPPED$LINK.TARGET$ADDR);
 82   2           CALL DISPLAY(.(SPACE, BAR, '$'));

 83   2           CALL LINE$OF$DASHES;
```

```
 84   2      END DISPLAY$INCOMING$LINK;

             /*** DISPLAY$UNSNAPPED$LINK DISPLAYS AN UNSNAPPED LINK OF THE CRT. ***/

 85   1      DISPLAY$UNSNAPPED$LINK : PROCEDURE (LINK$TYPE);

 86   2         DECLARE LINK$TYPE BYTE;

 87   2         CALL DISPLAY(.('     ', BAR, '    UNSNAPPED    ', BAR, '$'));
 88   2         CALL CRLF;

 89   2         IF LINK$TYPE = INCOMING THEN
 90   2            CALL DISPLAY(.('     ', BAR, '  INCOMING LINK ', BAR, '$'));

 91   2         ELSE CALL DISPLAY(.('     ', BAR, '  OUTGOING LINK ', BAR, '$'));

 92   2         CALL LINE$OF$DASHES;

 93   2      END DISPLAY$UNSNAPPED$LINK;

             /*** DISPLAY$SYM$NAME$TABLE DISPLAYS A DATA SYMBOLIC NAME TABLE (WHICH
                  WOULD BE STORED IN THE LINKAGE TABLE). ***/

 94   1      DISPLAY$SYM$NAME$TABLE : PROCEDURE (START$OF$TABLE, END$OF$TABLE);

 95   2         DECLARE START$OF$TABLE ADDRESS,
                        END$OF$TABLE ADDRESS,
                        SNT$PTR POINTER,
                        SNT BASED SNT$PTR STRUCTURE (
                           DESCRIPTOR BYTE,
                           LINK$OFFSET INTEGER,
                           ENTRY$POINT INTEGER,
                           NAME (1) BYTE),

                        I BYTE;
```

```
 96   2          SNT$PTR = START$OF$TABLE;

 97   2          CALL CRLF;
 98   2          CALL DISPLAY(.('     DATA SYMBOLIC NAME TABLE','$'));

 99   2          CALL PRINT$ADDRESS (START$OF$TABLE);
100   2          CALL CRLF;
101   2          CALL CRLF;

102   2          DO WHILE SNT$PTR < END$OF$TABLE;

103   3            CALL DISPLAY(.('          DESCRIPTOR  - ','$'));
104   3            CALL DISPLAY$HEX (SNT.DESCRIPTOR);
105   3            CALL CRLF;

106   3            CALL DISPLAY(.('          LINK OFFSET - ','$'));
107   3            CALL PRINT$VALUE (0, SNT.LINK$OFFSET);
108   3            CALL CRLF;

109   3            CALL DISPLAY(.('          ENTRY POINT - ','$'));
110   3            CALL PRINT$VALUE (0, SNT.ENTRY$POINT);
111   3            CALL CRLF;

112   3            CALL DISPLAY(.('          NAME        - ','$'));

113   3            DO I = 0 TO ((SNT.DESCRIPTOR AND 1FH) - 1);
114   4              CALL DISPLAY$CHAR (SNT.NAME (I));
115   4            END;

116   3            SNT$PTR = SNT$PTR + 5 + (SNT.DESCRIPTOR AND 1FH);
117   3            CALL CRLF;
118   3            CALL CRLF;

119   3          END;  /*  OF THE WHILE CLAUSE */
```

```
120   2           CALL CRLF;

121   2         END DISPLAY$SYM$NAME$TABLE;

                 /*** DISPLAY$A$LINKAGE$TABLE OUTPUTS A LINKAGE TABLE TO THE CRT ***/

122   1          DISPLAY$A$LINKAGE$TABLE : PROCEDURE (LINKAGE$TABLE$BASE);

123   2           DECLARE LINKAGE$TABLE$BASE POINTER,
                      TABLE BASED LINKAGE$TABLE$BASE STRUCTURE (
                        SIZE INTEGER,
                        SNT$ADDRESS ADDRESS,
                        BODY (1) BYTE),

                      LINK$BODY$PTR POINTER,
                      CHECK$BYTE BASED LINK$BODY$PTR BYTE;

124   2           CALL CRLF;

125   2           LINK$BODY$PTR = LINKAGE$TABLE$BASE + 4;

126   2           CALL DISPLAY(.('     LINKAGE TABLE  ','$'));
127   2           CALL OUTPUT$ADDR (0, COUNTER);
128   2           CALL DISPLAY (.('  (L',LITTLE$P,' = ','$'));
129   2           CALL PRINT$VALUE (0, LINKAGE$TABLE$BASE);
130   2           CALL DISPLAY$CHAR (')');
131   2           CALL CRLF;
132   2           CALL LINE$OF$DASHES;

133   2           CALL DISPLAY(.('      ', BAR, '  SIZE - ','$'));
134   2           CALL PRINT$VALUE (0, TABLE.SIZE);
135   2           CALL DISPLAY (.('  ', BAR, '$'));
136   2           CALL LINE$OF$DOTS;
```

```
 137   2         CALL DISPLAY (.('     ', BAR, '   SNT - ', '$'));
 138   2         CALL PRINT$VALUE (0, TABLE.SNT$ADDRESS);
 139   2         CALL DISPLAY (.('  ', BAR, '$'));
 140   2         CALL LINE$OF$DASHES;

                 /*** DISPLAY THE BODY OF THE LINKAGE TABLE ***/

 141   2         DO WHILE LINK$BODY$PTR < (LINKAGE$TABLE$BASE + TABLE.SIZE);

 142   3           IF CHECK$BYTE = 0 THEN DO;
 144   4             CALL DISPLAY$UNSNAPPED$LINK (INCOMING);
 145   4             LINK$BODY$PTR = LINK$BODY$PTR + 6;
 146   4           END;
                   ELSE
 147   3           IF CHECK$BYTE = JUMP$TO THEN DO;
 149   4             CALL DISPLAY$PROC$LINK (LINK$BODY$PTR);
 150   4             LINK$BODY$PTR = LINK$BODY$PTR + 5;
 151   4           END;
                   ELSE
 152   3           IF CHECK$BYTE = LOAD$LP THEN DO;
 154   4             CALL DISPLAY$INCOMING$LINK (LINK$BODY$PTR);
 155   4             LINK$BODY$PTR = LINK$BODY$PTR + 6;
 156   4           END;
                   IF CHECK$BYTE = LOAD$PTR THEN DO;
 159   4             CALL DISPLAY$DATA$LINK (LINK$BODY$PTR);
 160   4             LINK$BODY$PTR = LINK$BODY$PTR + 5;
 161   4           END;
                   ELSE
 162   3           IF CHECK$BYTE = PUSH$D THEN DO;
 164   4             CALL DISPLAY$UNSNAPPED$LINK (OUTGOING);
 165   4             LINK$BODY$PTR = LINK$BODY$PTR + 5;
 166   4           END;
                   ELSE
 167   3           IF TABLE.SNT$ADDRESS =  LINK$BODY$PTR THEN DO;
 169   4             CALL DISPLAY$SYM$NAME$TABLE (LINK$BODY$PTR,
```

```
                                               LINKAGE$TABLE$BASE + TABLE.SIZE);
 170   4            LINK$BODY$PTR = LINKAGE$TABLE$BASE + TABLE.SIZE;
 171   4          END;

                END;   /* OF THE WHILE LOOP */

 173   2        CALL CRLF;

 174   2      END DISPLAY$A$LINKAGE$TABLE;

              /*** OUTPUT$THE$LINK$TABLE DISPLAYS THE COMBINED LINKAGE TABLE ON
                   THE CRT.  IT DOES THIS BY SCANNING THE LINKAGE ADDRESS TABLE
                   AND OUTPUTING THE LINKAGE TABLE OF EACH VALID LINKAGE ADDRESS
                   TABLE ENTRY. ***/

 175   1      OUTPUT$THE$LINK$TABLE : PROCEDURE (LINK$ADDR$TABLE$BASE) PUBLIC;

 176   2        DECLARE LINK$ADDR$TABLE$BASE POINTER,
                        LINK$ADDR$TABLE BASED LINK$ADDR$TABLE$BASE (16) STRUCTURE (
                          VALID$BIT BYTE,
                          BASE$ADDR ADDRESS),

                        I BYTE;

 177   2        CALL DISPLAY$CHAR (FORM$FEED);
 178   2        CALL CRLF;
 179   2        CALL DISPLAY(.('     THE COMBINED LINKAGE TABLE','$'));
 180   2        CALL CRLF;
 181   2        CALL DISPLAY(.('     --------------------------','$'));
 182   2        CALL CRLF;
 183   2        CALL CRLF;

 184   2        DO I = 0 TO 15;

 185   3          IF LINK$ADDR$TABLE (I).VALID$BIT = VALID THEN DO;
```

```
 187   4               CALL DISPLAY$A$LINKAGE$TABLE (LINK$ADDR$TABLE (I).BASE$ADDR);
 188   4               COUNTER = COUNTER + 1;
 189   4             END;

 190   3          END;

 191   2       END OUTPUT$THE$LINK$TABLE;
 192   1      END DISLT;



MODULE INFORMATION:

     CODE AREA SIZE     = 0668H   1640D
     VARIABLE AREA SIZE = 001DH     29D
     MAXIMUM STACK SIZE = 0008H      8D
     387 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

```
PL/M-80 COMPILER    SYSTEM ROUTINES

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE COMMON
OBJECT MODULE PLACED IN :F1:COMMON.OBJ
COMPILER INVOKED BY:  PLM80 :F1:COMMON.SRC PAGELENGTH(38) TITLE('SYSTEM ROUTINES')


   1          COMMON : DO;

   2   1         DECLARE LIT LITERALLY 'LITERALLY',
                         DCI LIT 'DECLARE',
                         PROC LIT 'PROCEDURE',
                         ADDR LIT 'ADDRESS',
                         EXT LIT 'EXTERNAL',
                         SPACE LIT '20H',
                         TRUE  LIT '01H',
                         FALSE LIT '00H';

   3   1         DCI CHAR BYTE PUBLIC,
                     DECIMAL$BUFF (5) ADDR INITIAL(10000,1000,100,10,1),
                     FILE$BLK$ADDR ADDR INITIAL (5CH),
                     FILE$CONT$BLK BASED FILE$BLK$ADDR (33) BYTE;


   4   1         MON1 : PROC (A,B) EXT;            /*an explanation of MON1 and
   5   2          DCL A BYTE,                        and MON2 can be found on
                      B ADDR;                        page  196 of the thesis */
   6   2         END MON1;

   7   1         MON2 : PROC (A,B) BYTE EXT;
   8   2          DCI A BYTE,
                      B ADDR;
   9   2         END MON2;

  10   1         BOOT: PROC EXTERNAL;
  11   2         END BOOT;
```

```
                 /* READCHAR READS A CHARACTER FROM THE CONSOLE AND RETURNS THE
                    ASCII VALUE FOR THIS CHARACTER TO THE POINT OF CALL. IT ALSO
                    ASSIGNS THE ASCII VALUE OF THE CHARACTER TO THE PUBLIC
                    VARIABLE 'CHAR'.   */

  12   1         READCHAR : PROC BYTE PUBLIC;
  13   2          CHAR = MON2(1,0);
  14   2          RETURN CHAR;
  15   2         END READCHAR;

                 /* DISPLAY OUTPUTS TO THE CRT A CHARACTER STRING WHOSE ADDRESS IS
                    PASSED TO IT AS A PARAMETER.  THIS STRING MUST BE TERMINATED
                    BY THE ASCII CODE FOR A $.  NOTE THAT IF A '$' APPEARS IN THE
                    STRING TO BE OUTPUTED, DISPLAY WILL BE TERMINATED PREMATURELY.
                    A SAMPLE USE OF DISPLAY WOULD BE AS FOLLOWS:

                         CALL DISPLAY(.('THIS STRING WILL BE PRINTED.','$'));

                    */

  16   1         DISPLAY : PROC (A) PUBLIC;
  17   2          DCL A ADDR;
  18   2          CALL MON1(9,A);
  19   2         END DISPLAY;

                 /* PRINT OUTPUTS A CHARACTER STRING TO THE LINE PRINTER.  THE
                    FORMAT FOR PRINT IS THE SAME AS FOR DISPLAY.   */

  20   1         PRINT : PROC (A) PUBLIC;
  21   2          DCL A ADDR,
                      ITEM BASED A BYTE;

  22   2          DO WHILE ITEM <> '$';
  23   3           CALL  MON1(5,ITEM);
```

```
 24   3             A = A + 1;
 25   3            END;

 26   2           END PRINT;

                  /* CRLF CAUSES A CARRIAGE RETURN AND LINEFEED ON THE CRT. */

 27   1           CRLF: PROC PUBLIC;
 28   2            CALL DISPLAY(.(0DH,0AH,'$'));
 29   2           END CRLF;

 30   1           DISPLAY$ERROR : PROC (STRING$ADDR);
 31   2            DCL STRING$ADDR ADDR;

 32   2            CALL CRLF;
 33   2            CALL DISPLAY(STRING$ADDR);
 34   2            CALL BOOT;

 35   2           END DISPLAY$ERROR;

                  /* PAPER$ADVANCE CAUSES A CARRIAGE RETURN AND LINEFEED ON THE LINE
                     PRINTER. */

 36   1           PAPER$ADVANCE : PROC PUBLIC;
 37   2            CALL PRINT(.(0DH,0AH,'$'));
 38   2           END PAPER$ADVANCE;

                  /* DISPLAY$CHAR PRINTS A SINGLE CHARACTER ON THE CRT.  IT IS PASSED
                     THE ASCII CODE FOR THE CHARACTER TO BE DISPLAYED.  */

 39   1           DISPLAY$CHAR: PROC (CHARACTER) PUBLIC;
 40   2            DCL CHARACTER BYTE;
 41   2            CALL MON1(2,CHARACTER);
 42   2           END DISPLAY$CHAR;
```

```
                 /* PRINT$CHAR OUTPUTS A SINGLE CHARACTER TO THE LINE PRINTER. */

 43   1          PRINT$CHAR: PROC (CHARACTER) PUBLIC;
 44   2           DCL CHARACTER BYTE;
 45   2           CALL MON1(5,CHARACTER);
 46   2          END PRINT$CHAR;

                 /* OUTPUT$ADDR PRINTS A DECIMAL NUMBER ON EITHER THE CRT OR
                    THE LINE PRINTER DEPENDING ON THE 1ST PARAMETER IT IS PASSED
                    (0 FOR CRT, 1 FOR LPT).  THE SECOND PARAMETER IS THE SIGNED
                    ADDRESS VARIABLE TO BE DISPLAYED.  */

 47   1          OUTPUT$ADDR : PROC (DEVICE,VALUE) PUBLIC;
 48   2           DCL DEVICE BYTE,
                     VALUE ADDR,
                     (I,J) BYTE,
                     INTEGER$BUFF (6) BYTE,
                     COUNT BYTE,
                     FLAG BYTE;

 49   2              IF DEVICE > 1 THEN DEVICE = 0;
 51   2              FLAG = FALSE;
 52   2              J = 0;
 53   2              IF ROL(HIGH(VALUE),1) THEN DO;
 55   3               INTEGER$BUFF(0)='-';
 56   3               VALUE=-VALUE;
 57   3              END;
 58   2              ELSE INTEGER$BUFF(0)=SPACE;

 59   2              DO I=0 TO 4;
 60   3               COUNT = 30H;
 61   3               DO WHILE VALUE >= DECIMAL$BUFF(I);
 62   4                VALUE=VALUE-DECIMAL$BUFF(I);
 63   4                COUNT=COUNT+1;
 64   4                FLAG=TRUE;
```

```
65    4                END;
66    3                IF FLAG OR (I=4) THEN
67    3                 INTEGER$BUFF(J:=J+1)=COUNT;
                       ELSE
68    3                 INTEGER$BUFF(J:=J+1)=SPACE;
69    3               END;

70    2               DO CASE DEVICE;
71    3                DO;
72    4                 DO I=0 TO 5;
73    5                  IF INTEGER$BUFF(I) <> SPACE THEN
74    5                   CALL DISPLAY$CHAR(INTEGER$BUFF(I));
75    5                 END;
76    4                END;
77    3                DO;
78    4                 DO I=0 TO 5;
79    5                  IF INTEGER$BUFF(I) <> SPACE THEN
80    5                   CALL PRINT$CHAR(INTEGER$BUFF(I));
81    5                 END;
82    4                END;
83    3              END;    /* OF THE CASE STATEMENT */

84    2          END OUTPUT$ADDR;

                 /* OUTPUT$BYTE DISPLAYS A SIGNED BYTE VALUE AT EITHER THE CRT OR
                    LINE PRINTER.  */

85    1          OUTPUT$BYTE : PROC (DEVICE,VALUE) PUBLIC;
86    2           DCL DEVICE BYTE,
                      VALUE BYTE,
                      (I,J) BYTE,
                      INTEGER$BUFF (4) BYTE,
                      (COUNT,FLAG) BYTE;

87    2           IF DEVICE > 1 THEN DEVICE = 0;
```

```
  89   2            FLAG = FALSE;
  90   2            J = 0;
  91   2            IF ROL(VALUE,1) THEN DO;
  93   3             INTEGER$BUFF(0)='-';
  94   3             VALUE = -VALUE;
  95   3            END;
  96   2            ELSE INTEGER$BUFF(0)=SPACE;

  97   2            DO I = 2 TO 4;
  98   3             COUNT = 30H;
  99   3             DO WHILE VALUE >= DECIMAL$BUFF(I);
 100   4              VALUE=VALUE - DECIMAL$BUFF(I);
 101   4              COUNT = COUNT + 1;
 102   4              FLAG = TRUE;
 103   4             END;
 104   3             IF FLAG OR (I=4) THEN
 105   3              INTEGER$BUFF(J:=J+1)=COUNT;
                     ELSE
 106   3              INTEGER$BUFF(J:=J+1) = SPACE;
 107   3            END;
 108   2            DO CASE DEVICE;
 109   3             DO;     /* OUTPUT TO CRT */
 110   4              DO I=0 TO 3;
 111   5               IF INTEGER$BUFF(I) <> SPACE THEN
 112   5                CALL DISPLAY$CHAR(INTEGER$BUFF(I));
 113   5              END;
 114   4             END;

 115   3             DO;     /* OUTPUT TO LPT */
 116   4              DO I=0 TO 3;
 117   5               IF INTEGER$BUFF(I) <> SPACE THEN
 118   5                CALL PRINT$CHAR(INTEGER$BUFF(I));
 119   5              END;
 120   4             END;
 121   3            END;    /* OF THE CASE STATEMENT */
```

```
 122   2          END OUTPUT$BYTE;

                  /* SET$FILE$NAME LOADS A FILE TO BE OPERATED ON IN THE CPM
                     FILE$CONTROL$BLOCK.  THE NAME OF THE FILE IS DETERMINED BY THE
                     ADDRESS PASSED TO OPENFILE AS A PARAMETER.  THE FILENAME MUST BE
                     OF THE FORM FILENAME.FILETYPE.  THE FILENAME IS FROM ONE TO EIGHT
                     ALPHANUMERIC CHARACTERS WHILE THE FILETYPE IS FROM 0 TO THREE
                     ALPHANUMERIC CHARACTERS.  A SAMPLE USE OF THIS ROUTINE WOULD BE:

                                CALL OPENFILE(.(SAMPLE.ONE));

                   */

 123   1          SET$FILE$NAME : PROC (POINTER) PUBLIC;
 124   2           DCL POINTER ADDR,
                       CHARACTER BASED POINTER BYTE,
                       (I,J) BYTE;


 125   2           DO I=1 TO 11;
 126   3            FILE$CONT$BLK(I)=SPACE;
 127   3           END;

 128   2           I=0;
 129   2           DO WHILE (CHARACTER <> '.') AND (I < 9);
 130   3            FILE$CONT$BLK(I:=I+1) = CHARACTER;
 131   3            POINTER = POINTER + 1;
 132   3           END;
 133   2           IF I > 9 THEN CALL DISPLAY$ERROR(.('IMPROPER FILENAME','$'));
                    ELSE
 135   2             DO;
 136   3              I=8;
 137   3              POINTER=POINTER + 1;
 138   3              DO WHILE (CHARACTER <> SPACE) AND (I < 12);
```

```
 139   4                FILE$CONT$BLK(I:=I+1) = CHARACTER;
 140   4                POINTER = POINTER + 1;
 141   4               END;
 142   3              END;

 143   2           END SET$FILE$NAME;

                   /* DISPLAY$FCB DISPLAYS THE NAME IN THE FILE CONTROL BLOCK IF
                      THERE IS AN ERROR CONDITION IN OPEN OR CLOSE FILE. */

 144   1           DISPLAY$FCB : PROC;
 145   2             DCL NAME$BASE ADDR,
                           NAME BASED NAME$BASE (11) BYTE,
                           I BYTE;

 146   2             NAME$BASE = 5DH;

 147   2             CALL CRLF;
 148   2             CALL DISPLAY (.('THE FILE NAME IS : ','$'));
 149   2             DO I = 0 TO 10;
 150   3               CALL DISPLAY$CHAR (NAME(I));
 151   3             END;

 152   2          END DISPLAY$FCB;

                   /* OPENFILE OPENS THE FILE WHOSE NAME IS PASSED TO IT AS A FORMAL
                      PARAMETER.  THE FORMAT OF THE NAME IS DESCRIBED IN THE COMMENT
                      FOR SET$FILE$NAME.   */

 153   1           OPEN$FILE : PROC (POINTER) PUBLIC;
 154   2             DCL POINTER ADDR;

 155   2             CALL SET$FILE$NAME(POINTER);
```

```
 156   2             FILE$CONT$BLK(32)=0;
 157   2             FILE$CONT$BLK(0),FILE$CONT$BLK(12),FILE$CONT$BLK(15)=0;

 158   2             IF MON2(15,FILE$BLK$ADDR) = 255 THEN DO;
 160   3              CALL DISPLAY$FCB;
 161   3              CALL DISPLAY$ERROR(.('COULD NOT OPEN FILE','$'));
 162   3             END;

 163   2           END OPEN$FILE;

            /* CLOSEFILE CLOSES THE CURRENTLY OPENED FILE. */

 164   1           CLOSE$FILE : PROC PUBLIC;


 165   2            IF MON2(16,FILE$BLK$ADDR) = 255 THEN DO;
 167   3             CALL DISPLAY$FCB;
 168   3             CALL DISPLAY$ERROR(.('COULD NOT CLOSE FILE','$'));
 169   3            END;

 170   2           END CLOSE$FILE;

           /* READ$DISK READS A 128 BYTE BLOCK OF DATA FROM THE DISK AND
              LOADS IT INTO A BUFFER IN MEMORY WHOSE STARTING ADDRESS IS
              PASSED TO READ$DISK AS A FORMAL PARAMETER.  NOTE THAT BEFORE
              ONE CAN READ FROM A FILE ON DISK YOU MUST FIRST OPEN THE FILE.
              READ$DISK RETURNS A TRUE IF THE FILE WAS SUCCESSFULLY READ AND
              A FALSE IF THE END OF THE FILE WAS REACHED.  IT WILL TERMINATE
              PROGRAM EXECUTION IF AN ERROR IS DETECTED.   */

 171   1           READ$DISK : PROC (BUFFER$ADDR) BYTE PUBLIC;
 172   2            DCL BUFFER$ADDR ADDR,
                        TEMP BYTE;

 173   2            CALL MON1(26,BUFFER$ADDR);
```

```
174   2          TEMP = MON2(20,FILE$BLK$ADDR);

175   2          DO CASE TEMP;
176   3            RETURN TRUE;   /* FILE SUCCESSFULLY READ */
177   3            RETURN FALSE;  /* READ PAST END OF FILE  */
178   3            CALL DISPLAY$ERROR(.('FILE IMPROPERLY DEFINED','$'));
179   3          END;   /* OF CASE */

180   2         END READ$DISK;

                /* WRITE$DISK WRITES A 128 BYTE BLOCK OF DATA INTO A FILE.  NOTE
                  THAT THE CURRENT FILE AS DETERMINED BY EITHER AN OPEN$FILE
                  OR CREATE$FILE MUST BE THE ONE YOU DESIRE TO WRITE TO.
                  WRITE$DISK WILL COMMENCE WRITTING AT THE BEGINNING OF THE FILE
                  AND WILL DESTROY ANY EXISTING DATA AS IT WRITES.  THE DATA
                  WRITE$DISK WILL OUTPUT IS DETERMINED BY THE ADDRESS OF THE
                  128 BYTE BUFFER PASSED TO WRITE$DISK AS A FORMAL PARAMETER.
                  WRITE$DISK WILL RETURN A TRUE IF THE WRITE WAS SUCCESSFUL
                  OTHERWISE IT WILL TERMINATE PROGRAM EXECUTION IF AN ERROR
                  OCCURS.   */

181   1         WRITE$DISK : PROC (BUFFER$ADDR) BYTE PUBLIC;
182   2          DCL BUFFER$ADDR ADDR,
                     TEMP BYTE;

183   2          CALL MON1(26,BUFFER$ADDR);

184   2          TEMP = MON2(21,FILE$BLK$ADDR) AND 03H;

185   2          DO CASE TEMP;
186   3           RETURN TRUE;   /* WRITE WAS SUCCESSFUL */
187   3           CALL DISPLAY$ERROR(.('ERROR IN EXTENDING FILE','$'));
188   3           CALL DISPLAY$ERROR(.('DISK FULL','$'));
189   3           CALL DISPLAY$ERROR(.('DIRECTORY FULL','$'));
190   3          END;  /* OF CASE */
```

```
 191   2          END WRITE$DISK;

                  /* CREATE$FILE INITIALIZES A NEW FILE AS DETERMINED BY THE ADDRESS
                     OF THE FILENAME PASSED TO IT AS A FORMAI PARAMETER.  */

 192   1           CREATE$FILE : PROC (POINTER) PUBLIC;
 193   2             DCL POINTER ADDR;

 194   2             CALL SET$FILE$NAME(POINTER);

 195   2             IF MON2(22,FILE$BLK$ADDR) = 255 THEN
 196   2              CALL DISPLAY$ERROR(.('DIRECTORY FULL','$'));

 197   2          END CREATE$FILE;

                  /* DELETE$FILE DELETES A FILE AS DETERMINED BY THE ADDRESS OF THE
                     FILENAME PASSED TO IT AS A FORMAI PARAMETER.  */

 198   1          DELETE$FILE : PROC (POINTER) PUBLIC;
 199   2           DCL POINTER ADDR,
                         I BYTE;

 200   2           CALL SET$FILE$NAME(POINTER);

 201   2           I = MON2(19,FILE$BLK$ADDR);

 202   2          END DELETE$FILE;

 203   1       END COMMON;


MODULE INFORMATION:
```

```
     CODE AREA SIZE      = 05C5H   1477D
     VARIABLE AREA SIZE = 0040H     64D
     MAXIMUM STACK SIZE = 000AH     10D
     374 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

```
;  The System Routines invoke the CP/M operating system
; to perform their respective functions.  This entails
; calling the subroutines monitor_1 (mon1) and
; monitor_2 (mon2).  mon1 and mon2 very simply transfer
; control to the CP/M operating system via a jump vector
; located at 05H.  The pseudocode for mon1 and mon2 is
; as follows:
;
;  mon/mon2 : PROCEDURE (function_number, argument);
;
;      DECLARE function_number BYTE,
;              argument ADDRESS,
;
;         load the C register with function_number
;         load the D & E register with argument
;
;         jump to the CP/M entry point /* location 05H */
;
;         /* CP/M now performs the desired function as
;            determined by the function_number and
;            arguments  */
;
;         return byte value in the H & L reg   /* mon2 only */
;
;  end mon1
;

; the following is the assembly code for mon1 and mon2

ORG 0100H

CSEG            ;cseg tells the assembler to produce
                ;relocatable code

PUBLIC mon1, mon2

bdos equ 0005H

mon1 :          ;mon1 and mon2 are public labels
mon2 :

   JMP bdos

END 0100H
```

APPENDIX C - TEST PROGRAMS SOURCE LISTINGS

```
; DEMO displays a multiplication and addition table (in hex)
; of the numbers from 0 to 15

;  PROCEDURE Demo,
;
;     DECLARE Demo ENTRY POINT,
;             Mult PROCEDURE EXTERNAL,
;             Header DATA EXTERNAL,
;             Disply PROCEDURE EXTERNAL,
;
;             title_pointer : POINTER,
;             title ARRAY of BYTES BASED at title_pointer,
;
;     /* end of declarations */
;
;     PROCEDURE Add (number),
;
;       DECLARE number, i : BYTE,
;
;       FOR i = 0 to 15,
;         CALL Disply.Hex_value (i + number),
;       ENDFOR,
;
;     END Add,
;
;
;     PROCEDURE Build_table (routine),
;
;       DECLARE routine : PROCEDURE,
;               j : BYTE,
;
;       FOR j = 0 to 15,
;
;          CALL routine (i),
;          CALL Disply.Buffer (crlf),
;
```

Demo.object_code

```
;          ENDFOR,
;
;        END Build_table,
;
;        /* begin demo */
;
;        title_pointer = address of Header.title,
;        title = 'MULTIPLICATION',
;
;        CALL Disply.Buffer (header),
;        CALL Build_table (Mult),
;
;        title_pointer = address of Header.title,
;        title = 'ADDITION',
;
;        CALL Disply.Buffer (header),
;        CALL Build_table (Add),
;
;      END Demo,
;
;..........................................................

                ; assembly language program for Demo

0100            ORG 0100H

0100 C3EF01     JMP START

                ; DATA DECLARATIONS
```

Demo.object_code

```
0103                ROUTNE : DS 2
0105                TITPTR : DS 2
0107                NUMBER : DS 1
0108                I      : DS 1
0109                J      : DS 1

000D =              CR     : EQU 0DH
000A =              LF     : EQU 0AH
0026 =              DELIM  : EQU '&'

012A 0D0A26         CRLF   : DB CR, LF, '&'
010D 4D554C5449     MTITLE : DB 'MULTIPLICATION&'
011C 4144444954     ATITLE : DB 'ADDITION       &'

                  ; Add displays the sum of number and 0 through 15

                  PADD :

012B 210701          LXI H, NUMBER       ;load the H & L regs w/ the address of number
012E 73              MOV M, E            ;move the parameter into number

012F 210801          LXI H, I            ;load the H & L regs w/ the address of i
0132 3600            MVI M, 0            ;initialize i to 0
                  LOOP1 :

0134 3E0F            MVI A, 15           ;load 15 into the accumulator
0136 210801          LXI H, I            ;load the H & L regs w/ the address of i
0139 BE              CMP M               ;compare i and 15
013A DA5801          JC ENDFR1           ;jump to endfor if i > 15
```

Demo.object_code

```
013D 210801        LXI H, I          ;load the H & I regs w/ the address of i
0140 7E            MOV A, M          ;move i into the accumulator
0141 210701        LXI H, NUMBER     ;load the H & I regs w/ the address of number
0144 86            ADD M             ;add number to i
0145 5F            MOV E, A          ;move the result into the E reg

                   ; dynamically link and call disply.hex_value

0146 C5            PUSH B            ;save the linkage pointer
0147 215001        LXI H, RETAD1     ;save the return address on the stack
014A E5            PUSH H
014B 211900        LXI H, 19H        ;load the offset of the outgoing link
014E 09            DAD B             ;compute Lp + outgoing link offset
014F E9            PCHL              ;jump to the outgoing link

                 RETAD1 :
0150 C1            POP B             ;restore the linkage pointer

0151 210801        LXI H, I          ;load the H & I regs w/ the address of i
0154 34            INR M             ;i = i + 1
0155 C33401        JMP LOOP1         ;jump to loop

0158 C9          ENDPR1 : RET        ;end of sum

                 ;.....................................................

                 ; PROCEDURE Build_table (routine)

                 BLDTBL :
0159 EB           XCHG                ;load the parameter into the H & I regs
```

Demo.object_code

```
015A 220301       SHID ROUTNE          ;and store the address of the parameter in routine
015D 210901       IXI H, J             ;load the H & L regs w/ the address of j
0160 3600         MVI M, 0             ;initialize j to 0

                LOOP2 :
0162 3E0F         MVI A, 15            ;load 15 into the accumulator
0164 210901       IXI H, J             ;load the H & L regs w/ the address of j
0167 BE           CMP M                ;compare i and 15
0168 DA8E01       JC ENDFR2            ;jump to endfor if j > 15

                  ;call routine (j)

016B 210901       IXI H, J             ;load the H & L regs w/ the address of j
016E 5E           MOV E, M             ;move i into the E reg
016F C5           PUSH B               ;save the linkage pointer
0170 217801       IXI H, RETAD2        ;save the return address on the stack
0173 E5           PUSH H
0174 2A0301       LHID ROUTNE          ;load the H & L regs w/ the address of routine
0177 E9           PCHI                 ;jump to routine

                RETAD2 :
0178 C1             POP B              ;restore the linkage pointer

                    ;dynamically link and call disply.buffer (crlf)

0179 110A01         LXI D, CRLF        ;load the D & E regs w/ the address of crlf
017C C5             PUSH B             ;save the linkage pointer
017D 218601         LXI H, RETAD3      ;save the return address on the stack
0180 E5             PUSH H
0181 211E00         LXI H, 1EH         ;load the offset of the outgoing link
```

```
0184 09          DAD B              ;compute Lp + outgoing link offset
0185 E9          PCHI               ;jump to the outgoing link

               RETAD3 :
0186 C1          POP B              ;restore the linkage pointer
0187 210901      LXI H, J           ;load the  H & L regs w/ the address of j
018A 34          INR M              ;j = j + 1
018B C36201      JMP LOOP2          ;jump to loop2

018E C9        ENDFR2 : RET         ;end of build_table

               ;..................................................

               ;   /* begin demo */

               START :

                 ; dynamically link to header.title

018F C5          PUSH B             ;save the linkage pointer
0190 219901      LXI H, RETAD4      ;save the return address on the stack
0193 E5          PUSH H
0194 210F00      LXI H, 0FH         ;load the offset of the outgoing link
0197 09          DAD B              ;compute Lp + outgoing link offset
0198 E9          PCHI               ;jump to the outgoing link

               RETAD4 :
0199 C1          POP B              ;restore the linkage pointer
019A EB          XCHG               ;move the address of header.title into H & L regs
019B 220501      SHLD TITPTR        ;store header.title into title_pointer
```

```
019E 110D01        LXI D, MTITLE        ;load the D & E regs w/ the address of mtitle
01A1 2A0501        LHLD TITPTR          ;load the H & L regs w/ title_pointer
               LOOP3 :
01A4 1A            LDAX D               ;load the accumulator w/ a character from mtitle
01A5 FE26          CPI DELIM            ;is that character the delimiter
01A7 CAB001        JZ ENDLP1            ;if so, jump to endloop1
01AA 77            MOV M, A             ;otherwise store the character in header.title
01AB 23            INX H                ;increment title_pointer
01AC 13            INX D                ;increment the address of mtitle
01AD C3A401        JMP LOOP3            ;and continue in loop

               ENDLP1 :

               ; dynamically link to header

01B0 C5              PUSH B             ;save the linkage pointer
01B1 21BA01          LXI H, RETAD5      ;save the return address on the stack
01B4 E5              PUSH H
01B5 211400          LXI H, 14H         ;load the offset of the outgoing link
01B8 09              DAD B              ;compute Lp + offset of outgoing link
01B9 E9              PCHL               ;jump to the outgoing link

               RETAD5 :
01BA C1              POP B              ;restore the linkage pointer

               ; dynamically link and call disply.buffer (header). the address of
               ; header is in the D & E regs

01BB C5              PUSH B             ;save the linkage pointer
01BC 21C501          LXI H, RETAD6      ;save the return address on the stack
```

Demo.object_code

```
 01BF E5              PUSH H
 01C0 211E00          LXI H, 1EH          ;load the offset of the outgoing link
 01C3 09              DAD B               ;compute Lp + offset of outgoing link
 01C4 E9              PCHL                ;jump to the outgoing link

                 RETAD6 :
 01C5 C1              POP B

 01C6 210A00          LXI H, 0AH          ;load the offset of the outgoing link
 01C9 09              DAD B               ;compute Lp + offset for Mult
 01CA EB              XCHG                ;store outgoing link address for Mult
                                          ;in the D & E regs
 01CB CD5901          CALL BLDTBL         ;and call build_table

                      ; dynamically link to header.title

 01CE C5              PUSH B              ;save the linkage pointer
 01CF 21D801          LXI H, RETAD7       ;save the return address on the stack
 01D2 E5              PUSH H
 01D3 210F00          LXI H, 0FH          ;load the offset of the outgoing link
 01D6 09              DAD B               ;compute Lp + outgoing link offset
 01D7 E9              PCHL                ;jump to the outgoing link

                 RETAD7 :
 01D8 C1              POP B               ;restore the linkage pointer
 01D9 EB              XCHG                ;move the address of header.title into H & L regs
 01DA 227501          SHLD TITPTR         ;store header.title address into title_pointer
 01DD 111C01          LXI D, ATITLE       ;load the D & E regs w/ the address of atitle
 01E0 2A0521          LHLD TITPTR         ;load the H & L regs w/ title_pointer
                 LOOP4 :
```

Demo.object_code

```
01E3 1A          LDAX D            ;load the accumulator w/ a character from atitle
01E4 FE26        CPI DELIM         ;is that character the delimiter
01E6 CAEF01      JZ ENDLP2         ;if so, jump to endloop2
01E9 77          MOV M, A          ;otherwise store the character in header.title
01EA 23          INX H             ;increment title_pointer
01EB 13          INX D             ;increment the address of atitle
01EC C3E301      JMP LOOP4         ;and continue in loop

              ENDLP2 :

              ; dynamically link to header

01EF C5          PUSH B            ;save the linkage pointer
01F0 21F901      LXI H, RETAD8     ;save the return address on the stack
01F3 E5          PUSH H
01F4 211400      LXI H, 14H        ;load the offset of the outgoing link
01F7 09          DAD B             ;compute Lp + outgoing link offset
01F8 E9          PCHL              ;jump to the outgoing link

              RETAD8 :
01F9 C1          POP B             ;restore the linkage pointer

              ; dynamically link and call disply.buffer (header), the address of
              ; header is in the D & E regs

01FA C5          PUSH B            ;save the linkage pointer
01FB 210402      LXI H, RETAD9     ;save the return address on the stack
01FE E5          PUSH H
01FF 211E00      LXI H, 1EH        ;load the offset of the outgoing link
0202 09          DAD B             ;compute Lp + outgoing link offset
```

Demo.object_code

```
0203 E9              PCHL                    ;jump to the outgoing link

                 RETAD9 :
0204 C1              POP B                   ;restore the linkage pointer

0205 212B01          LXI H, PADD             ;load the H & L regs w/ the address of add
0208 EB              XCHG                    ;move the address of add into the D & E regs
0209 CD5901          CALL BLDTBL             ;and call build_table

020C C9              RET                     ;end of demo

                 ;..........................................................

                 ;symbolic name table

                 ;entry point into demo

020D 04             DESC0  : DB 04
020E 0400           LINK0  : DB 04, 00
0210 8E00           ENTRY0 : DB 8EH, 00
0212 44454D4F       NAME0  : DB 'DEMO'

                 ;entry for mult

0216 04             DESC1  : DB 04H
0217 0A00           LINK1  : DB 0AH, 00
0219 0000           ENTRY1 : DB 00, 00
021B 4D554C54       NAME1  : DB 'MULT'

                 ;entry for header.title
```

Demo.object_code

```
021F 8C            DESC2  : DB 8CH
0220 0F00          LINK2  : DB 0FH, 00
0222 0000          ENTRY2 : DB 00, 00
0224 4845414445    NAME2  : DB 'HEADER:TITLE'

                   ;entry for header

0230 86            DESC3  : DB 86H
0231 1400          LINK3  : DB 14H, 00
0233 0000          ENTRY3 : DB 00, 00
0235 4845414445    NAME3  : DB 'HEADER'

                   ;entry for display.hex_value

023B 10            DESC4  : DB 10H
023C 1900          LINK4  : DB 19H, 00
023E 0000          ENTRY4 : DB 00, 00
0240 444953504C    NAME4  : DB 'DISPLY:HEX_VALUE'

                   ;entry for disply.buffer

0250 0D            DESC5  : DB 0DH
0251 1E00          LINK5  : DB 1EH, 00
0253 0000          ENTRY5 : DB 00, 00
0255 444953504C    NAME5  : DB 'DISPLY:BUFFER'

                   ;end of symbolic name table

0262               END 0100H
```

```
                 ; this is the template for demo

0100             ORG 0100H

0100 2300        SIZE : DB 35, 00
0102 0D01        SNT  : DB 0DH, 01H
                 BODY :

0104 000000000000  DB 00, 00, 00, 00, 00, 00   ;incoming link into demo

010A D5            PUSH D                      ;outgoing link for mult
010B 110900        LXI D, 09
010E E7            RST 4

010F D5            PUSH D                      ;outgoing link for header.title
0110 111200        LXI D, 18
0113 E7            RST 4

0114 D5            PUSH D                      ;outgoing link for header
0115 112300        LXI D, 35
0118 E7            RST 4

0119 D5            PUSH D                      ;outgoing link for disply.hex_value
011A 112E00        LXI D, 46
011D E7            RST 4

011E D5            PUSH D                      ;outgoing link for disply.buffer
011F 114300        LXI D, 67
0122 E7            RST 4

0123             END 0100H
```

```
                   ; this is the relocation bits file for demo

0100               ORG 0100H

0100 2400          SIZE  : DB 36, 00
0102 4000          L0100 : DB 01000000B, 00000000B
0104 0000          L0110 : DB 00000000B, 00000000B
0106 0008          L0120 : DB 00000000B, 00001000B
0108 8112          L0130 : DB 10000001B, 00010010B
010A 2080          L0140 : DB 00100000B, 10000000B
010C 2212          L0150 : DB 00100010B, 00010010B
010E 0448          L0160 : DB 00000100B, 01001000B
0110 4422          L0170 : DB 01000100B, 00100010B
0112 0088          L0180 : DB 00000000B, 10001000B
0114 4009          L0190 : DB 01000000B, 00001001B
0116 2082          L01A0 : DB 00100000B, 10000010B
0118 2004          L01B0 : DB 00100000B, 00000100B
011A 0008          L01C0 : DB 00000000B, 00001000B
011C 8012          L01D0 : DB 10000000B, 00010010B
011E 4104          L01E0 : DB 01000001B, 00000100B
0120 4008          L01F0 : DB 01000000B, 00001000B
0122 0220          L0200 : DB 00000010B, 00100000B

0124               END 0100H
```

```
                ; this is the header for the table generated by demo

0100            ORG 0100H

000D =          CR       : EQU 0DH
000A =          LF       : EQU 0AH
0026 =          DELIM    : EQU '&'

                HEADER   :

0100 0D0A0D0A      DB CR, LF, CR, LF
0104 2020202020    DB '                 '

                TITLE1   :

0111 2020203134    DB '   14 spaces   '
011F 205441424C    DB ' TABLES '
0127 0D0A          DB CR, LF
0129 2020202020    DB '                 '
0136 2D2D2D2D2D    DB '---------------------'
014B 0D0A0D0A0D    DB CR, LF, CR, LF, CR, LF

0151 2030202031    DB '0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F'
0181 0D0A0D0A      DB CR, LF, CR, LF
0185 26            DB DELIM

                ; end of header

0186            END 0100H
```

```
                 ; this is the template for header

0100             ORG 0100H

0100 1900        SIZE : DB 25, 00
0102 0400        SNT  : DB 04, 00

                 ; symbolic name table for header

                 ;entry point for header

0104 06            DESC0  : DB 06
0105 0000          LINK0  : DB 00, 00
0107 0000          ENTRY0 : DB 00, 00
0109 4845414445    NAME0  : DB 'HEADER'

                 ;entry point for header.title

010F 05            DESC1  : DB 05
0110 0000          LINK1  : DB 00, 00
0112 1100          ENTRY1 : DB 11H, 00
0114 5449544C45    NAME1  : DB 'TITIE'

                 ; end of symbolic name table

0119             END 0100H
```

```
; Mult displays the product of number and 0 through 15 on the
; CRT

;   PROCEDURE Mult (number),
;
;     DECLARE number, i : BYTE,
;
;     FUNCTION Product (x, y).
;
;       DECLARE  x, y : BYTE,
;                sum, j : BYTE,
;
;       sum = 0,
;
;       FOR j = 1 to x,
;         sum = sum + y,
;       ENDFOR,
;
;       RETURN sum,
;
;     END Product,
;
;     /* begin mult */
;
;     FOR i = 0 to 15,
;       CALL Disply.hex_value (Product (i, number)),
;     ENDFOR,
;
;   END Mult,
;
;..........................................................

               ;  assembly language program for Mult

0100           ORG 0100H
```

Mult.object_code

```
0100 C34101     JMP START

                ; DATA DECLARATIONS

0103            I         : DS 1
0104            J         : DS 1
0105            SUM       : DS 1
0106            X         : DS 1
0107            Y         : DS 1
0108            NUMBER    : DS 1
0109            PARAMS    : DS 2

                ; product multiplies two number by repeated addition

                PRODCT :

010B 210901       LXI H, PARAMS      ;load the H & L regs w/ the address of parameters
010E 7E           MOV A, M           ;move the first parameter into the accumulator
010F 320601       STA X              ;store the first parameter into x
0112 23           INX H              ;increment the H & L regs to point to the second
                                     ;parameter
0113 7E           MOV A, M           ;move the second parameter into the accumulator
0114 320701       STA Y              ;store that parameter into y

0117 3E00         MVI A, 0           ;move 0 into the accumulator
0119 320501       STA SUM            ;and initialize sum to 0
011C 3E01         MVI A, 1           ;move 1 into the accumulator
011E 320401       STA J              ;and initialize j to 1

                LOOP1 :
```

Mult.object_code

```
0121 3A0601      LDA X              ;load the accumulator with x
0124 210401      LXI H, J           ;load the H & I regs w/ the address of j
0127 BE          CMP M              ;compare j to the value of x
0128 DA3C01      JC ENDFR1          ;jump out of the loop if x < j
012B 3A0501      LDA SUM            ;otherwise move sum into the accumulator
012E 210701      LXI H, Y           ;load the H & I regs w/ the address of y
0131 86          ADD M              ;and add y to sum
0132 320501      STA SUM            ;store the result in sum
0135 210401      LXI H, J           ;load the H & I regs w/ the address of j
0138 34          INR M              ;j = j + 1
0139 C32101      JMP LOOP1          ;and jump to loop1

               ENDFR1 :
013C 210501      LXI H, SUM         ;load the H & I regs w/ the address of sum
013F 5E          MOV E, M           ;move sum into the E reg
0140 C9          RET                ;return sum to point of call, end of product

               START :

                 ; procedure mult

0141 210801      LXI H, NUMBER      ;load the H & I regs w/ the address of number
0144 73          MOV M, E           ;move the parameter into number

0145 3E00        MVI A, 0           ;load 0 into the accumulator
0147 320301      STA I              ;initialize i to 0

               LOOP2 :
```

Mult.object_code

```
 014A 3E0F         MVI A, 15          ;move 15 into the accumulator
 014C 210301       LXI H, I           ;load the H & L regs w/ the address of i
 014F BE           CMP M              ;compare i to 15
 0150 DA7401       JC ENDFR2          ;jump to endfor if i > 15

                   ;load the parameters i and number into params

 0153 210901       LXI H, PARAMS      ;load the H & L regs w/ the address of params
 0156 3A0301       LDA I              ;load i into the accumulator
 0159 77           MOV M, A           ;move i into the first parameter
 015A 23           INX H              ;increment the address of params
 015B 3A0801       LDA NUMBER         ;load number into the accumulator
 015E 77           MOV M, A           ;move number into the second parameter
 015F CD0B01       CALL PRODCT

                 ; dynamically link and call disply.hex_value, the value
                 ; product (i, number) is in the E reg

 0162 C5           PUSH B             ;save the linkage pointer
 0163 216C01       LXI H, RETAD1      ;save the return address on the stack
 0166 E5           PUSH H
 0167 210A00       LXI H, 0AH         ;load the offset of the outgoing link
 016A 09           DAD B              ;compute lp + outgoing link offset
 016B E9           PCHL               ;jump to the outgoing link

                 RETAD1 :
 016C C1             POP B            ;restore the linkage pointer
 016D 210301       LXI H, I           ;load the H & L regs w/ the address of i
 0170 34           INR M              ;i = i + 1
 0171 C34A01       JMP LOOP2          ;and jump to loop2
```

Mult.object_code

```
0174 C9          ENDER2 : RET             ;end of mult

                 ;...........................................................

                 ;symbolic name table

                 ;entry point into mult

0175 04             DESC0  : DB 04
0176 0400           LINK0  : DB 04, 00
0178 4100           ENTRY0 : DB 41H, 00
017A 4D554C54       NAME0  : DB 'MULT'

                 ;entry for disply.hex_value

017E 10             DESC1  : DB 10H
017F 0A00           LINK1  : DB 0AH, 00
0181 0000           ENTRY1 : DB 00, 00
0183 444953504C     NAME1  : DB 'DISPLY:HEX_VALUE'

                 ;end of symbolic name table

0193             END 0100H
```

```
                  ; this is the template for mult

0100              ORG 0100H

0100 0F00         SIZE : DB 15, 00
0102 7500         SNT  : DB 75H, 00
                  BODY :

0104 000000000000   DB 00, 00, 00, 00, 00, 00   ;incoming link into mult

010A D5             PUSH D                      ;outgoing link for disply.hex_value
010B 110900         LXI D, 09H
010E E7             RST 4

010F              END 0100H
```

```
                    ; this is the relocation bits file for mult

0100                ORG 0100H

0100 1100           SIZE  : DB 17, 00
0102 4008           L0100 : DB 01000000B, 00001000B
0104 8421           L0110 : DB 10000100B, 00100001B
0106 2449           L0120 : DB 00100100B, 01001001B
0108 1224           L0130 : DB 00010010B, 00100100B
010A 2084           L0140 : DB 00100000B, 10000100B
010C 4908           L0150 : DB 01001001B, 00001000B
010E 8802           L0160 : DB 10001000B, 00000010B
0110 20             L0170 : DB 00100000B

0111                END 0100H
```

```
; Display outputs either a byte value in hexidecimal form
; (Hex_value) or an ASCII character string (Buffer)
;
;   PROCEDURE Disply,
;
;      DECLARE Hex_value ENTRY POINT,
;              Buffer ENTRY POINT,
;
;      /* end of declarations */
;
;      /* Print displays an ASCII byte on the CRT */
;
;         PROCEDURE Print (ascii_byte),
;            DECLARE ascii_byte : BYTE,
;              OUTPUT (ascii_byte),
;         END Print,
;
;      /* Hex_value prints the hexidecimal value of the
;         parameter 'a_byte' on the CRT */
;
;      PROCEDURE Hex_value (a_byte),
;
;         DECLARE a_byte, temp : BYTE,
;
;         /* Print_hex displays the hex value of a nibble
;            on the CRT */
;
;         PROCEDURE Print_hex (nibble),
;
;           /* if nibble is less then 10, then print a digit,
;              otherwise print the hex value A,B,C,D,E, or F */
;
;           IF nibble < 10 then CALL Print (nibble + 30H),
;           ELSE CALL Print (nibble + 37H);
```

Disply.object_code

```
;
;        END Print_hex,
;
;        /* begin hex_value */
;
;        temp = SHIFT_RIGHT_4 (a_byte AND F0H),
;        CALL Print_hex (temp),
;
;        CALL Print_hex (a_byte AND 0FH),
;
;        CALL Print (space),
;
;     END Hex_value,
;
;     /* Buffer displays the contents of an ASCII string
;        on the CRT */
;
;     PROCEDURE Buffer (string_pointer),
;
;        DECLARE string_pointer : POINTER,
;                string_byte BYTE BASED at string_pointer,
;
;        DO WHILE string_byte <> delimiter,
;          CALL Print (string_byte),
;          string_pointer = string_pointer + 1,
;        ENDWHILE,
;
;     END Buffer,
;
;     END Disply,
```

Disply.object_code

```
        ;
        ;..........................................................
                  ; assembly language program for Disply

0100              ORG 100H

0100 C31501       JMP START

                  ; DATA DECLARATIONS

0103              NIBBLE : DS 1
0104              ABYTE  : DS 1
0105              TEMP   : DS 1
0106              STRPTR : DS 2
0108              SBYTE  : DS 1

0026 =            DELIM  : EQU '&'

                  ; Print outputs the contents of the E register to the CRT

                  PRINT:

0109 E5              PUSH H        ;save the registers
010A C5              PUSH B
010B F5              PUSH PSW

010C 0E02            MVI C, 02H    ;tell the operating system to print
010E CD0500          CALL 05H      ;call the opsys print routine
```

Disply.object_code

```
0111 F1            POP PSW       ;restore the registers
0112 C1            POP B
0113 E1            POP H
0114 C9            RET

                START :

                ; PROCEDURE Print_hex

                PRTHEX :
0115 210301        LXI H, NIBBLE      ;load the H & L regs w/ the address of nibble
0118 73            MOV M, E           ;move the parameter into nibble

0119 7E            MOV A, M           ;move nibble into the accumulator
011A FE0A          CPI 10             ;compare nibble to 10
011C D22C01        JNC LABEL1         ;if nibble >= 10 then jump to label1

011F 210301        LXI H, NIBBLE      ;load the H & L regs w/ the address of nibble
0122 3E30          MVI A, 30H         ;move 30H into the accumulator
0124 86            ADD M              ;add nibble to 30H
0125 5F            MOV E, A           ;move the result into the E reg
0126 CD0901        CALL PRINT         ;and call print
0129 C33601        JMP LABEL2         ;skip the ELSE clause

                LABEL1 :
012C 210301        LXI H, NIBBLE      ;load the H & L regs w/ the address of nibble
012F 3E37          MVI A, 37H         ;move 37H into the accumulator
0131 86            ADD M              ;add nibble to 37H
0132 5F            MOV E, A           ;move the result into the E reg
```

Disply.object_code

```
0133 CD0901         CALL PRINT          ;and call print

0136 C9          LABEL2 : RET           ;return to the point of call


                 ;.........................................................

                 ; PROCEDURE Hex_value (a_byte),

                 HEXVAL :
0137 210401         LXI H, ABYTE        ;load the H & L regs w/ the address of a_byte
013A 73             MOV M, E            ;move the actual parameter into a_byte

013B 7E             MOV A, M            ;move a_byte into the accumulator
013C E6F0           ANI 0F0H            ;AND a_byte with F0H
013E 0F             RRC                 ;shift the result right 4 bits
013F 0F             RRC
0140 0F             RRC
0141 0F             RRC
0142 210501         LXI H, TEMP         ;load the H & L regs w/ the address of temp
0145 77             MOV M, A            ;move the result into temp
0146 5E             MOV E, M            ;move temp into the E reg
0147 CD1501         CALL PRTHEX         ;and call Print_hex

014A 210401         LXI H, ABYTE        ;load the H & L regs w/ the address of a_byte
014D 7E             MOV A, M            ;load the accumulator with a_byte
014E E60F           ANI 0FH;            ;AND a_byte with 0FH
0150 5F             MOV E, A            ;move the result into the E reg
0151 CD1501         CALL PRTHEX         ;and call Print_hex
```

Disply.object_code

```
0154 1E20        MVI E, 20H          ;move ASCII space into the E reg
0156 CD0901      CALL PRINT          ;and call Print

0159 C9          RET                 ;end of Hex_value

                ;..................................................

                ; PROCEDURE Buffer (string_pointer)

                BUFFER :
015A EB          XCHG                 ;move the parameter into the H & L regs
015B 220601      SHLD STRPTR          ;and store it in string_pointer

                WHILE :
015E 2A0601      LHLD STRPTR          ;load string_pointer into the H & L regs
0161 7E          MOV A, M             ;move string_byte into the accumulator
0162 FE26        CPI DELIM            ;compare string_byte with the delimiter
0164 CA7501      JZ ENDWHL            ;jump to ENDWHILE if string_byte = delimiter
0167 5E          MOV E, M             ;else move string_byte into the E reg
0168 CD0901      CALL PRINT           ;and call Print
016B 2A0601      LHLD STRPTR          ;load string_pointer into the H & L regs
016E 23          INX H                ;increment string_pointer
016F 220601      SHLD STRPTR          ;and store the result
0172 C35E01      JMP WHILE            ;continue in the WHILE loop

0175 C9         ENDWHL : RET          ;end of Buffer

                ;..................................................

                ;symbolic name table
```

```
Disply.object_code


                    ;entry point for Hex_value

 0176 09              DESC0   : DB 09
 0177 0400            LINK0   : DB 04, 00
 0179 3700            ENTRY0  : DB 37H, 00
 017B 4845585F56      NAME0   : DB 'HEX_VALUE'

                    ;entry point for Buffer

 0184 06              DESC1   : DB 06
 0185 0A00            LINK1   : DB 10, 00
 0187 5A00            ENTRY1  : DB 5AH, 00
 0189 4255464645      NAME1   : DB 'BUFFER'

                    ;end of symbolic name table

                    ;end of Disply

 018F               END 0100H
```

```
                ; this is the template for disply

0100            ORG 0100H

0100 1000       SIZE : DB 16, 00
0102 7600       SNT  : DB 76H, 00
                BODY :

0104 0000000000  DB 00, 00, 00, 00, 00, 00   ;incoming link for hex_value

010A 0000000000  DB 00, 00, 00, 00, 00, 00   ;incoming link for buffer

0110            END 0100H
```

```
                 ; this is the relocation bits file for disply

0100             ORG 0100H

0100 1100        SIZE  : DB 17, 00

0102 4000        L0100 : DB 01000000B, 00000000B
0104 0204        L0110 : DB 00000010B, 00000100B
0106 8124        L0120 : DB 10000001B, 00100100B
0108 0880        L0130 : DB 00001000B, 10000000B
010A 1090        L0140 : DB 00010000B, 10010000B
010C 2109        L0150 : DB 00100001B, 00001001B
010E 0448        L0160 : DB 00000100B, 01001000B
0110 90          L0170 : DB 10010000B

0111             END 0100H
```

Sum.object_code

```
; SUM adds the bytes of the external data structure ARRAY
; and displays the result on the CRT

;  PROCEDURE Sum,
;
;     DECLARE Sum ENTRY POINT,
;             Array DATA EXTERNAL,
;             Disply PROCEDURE EXTERNAL,
;
;             result : BYTE,
;             array_pointer : POINTER,
;             data_array BASED at array_pointer STRUCTURE of
;               number_of_bytes : BYTE,
;               data : ARRAY of BYTES,
;               END,
;
;             i : BYTE,
;
;     /* end of declarations */
;
;     array_pointer = address of array,
;     result = 0,
;
;     FOR i = 1 to data_array.number_of_bytes,
;        result = result + data_array.data (i),
;     ENDFOR,
;
;     CALL disply.buffer ('The sum of the data array is ','&'),
;     CALL disply.hex_buffer (result),
```

Sum.object_code

```
        ;
        ;     /* generate a carriage return and line feed */
        ;
        ;     CALL disply.buffer (CR, LF, '&'),
        ;     CALL disply.buffer ('End of Sum','&');
        ;
        ;   END Sum.

                   ; assembly language program for Sum

0100               ORG 0100H

0100 C33401        JMP START

                   ; DATA DECLARATIONS

0103                 RESULT : DS 1
0104                 I      : DS 1
0105                 POINTR : DS 2

000D =               CR     : EQU 0DH
000A =               LF     : EQU 0AH

0107 5448452053      HEADER : DB 'The sum of the data array is &'
0125 454E44204F      ENDING : DB 'End of Sum &'
0131 0D0A26          CRLF   : DB CR, LF, '&'

                   START :

                   ; dynamically link to array to get the value of array_pointer
```

Sum.object_code

```
0134 C5             PUSH B              ;save the linkage pointer
0135 213E01         LXI H, RETAD1       ;save the return address on the stack
0138 E5             PUSH H
0139 210A00         LXI H, 000AH        ;load the outgoing link offset in the H & L regs
013C 09             DAD B               ;compute Lp + outgoing link offset
013D E9             PCHL                ;jump to outgoing link

                RETAD1 :

013E C1             POP B               ;restore the linkage pointer
013F EB             XCHG                ;move array_pointer into the H & L regs
0140 220501         SHLD POINTR         ;store array_pointer
0143 3E00           MVI A, 0            ;set the accumulator to 0
0145 210301         LXI H, RESULT       ;load the H & L regs w/ the address of result
0148 77             MOV M, A            ;initialize result to 0
0149 210401         LXI H, I            ;load the H & L regs with the address of i
014C 3601           MVI M, 1            ;initialize i to 1

                LOOP :

014E 2A0501         LHLD POINTR         ;load the  H & L regs with array_pointer
0151 7E             MOV A, M            ;move number_of_bytes into the accumulator
0152 210401         LXI H, I            ;load the H & L regs w/ the address of i
0155 BE             CMP M               ;compare i and number_of_bytes
0156 DA7301         JC ENDFOR           ;jump to endfor if i > number_of_bytes

0159 210301         LXI H, RESULT       ;load the H & L regs w/ the address of result
015C 7E             MOV A, M            ;move result into the accumulator
015D 210401         LXI H, I            ;load the H & L regs w/ the address of i
```

Sum.object_code

```
0160 5E          MOV E, M           ;move i into the E reg
0161 1600        MVI D, 0           ;clear the D reg
0163 2A0501      LHLD POINTR        ;move array_pointer into the H & L regs
0166 19          DAD D              ;compute the address of data_array.data (i)
0167 86          ADD M              ;add data_array.data (i) to result
0168 210301      LXI H, RESULT      ;load the H & L regs w/ the address of result
016B 77          MOV M, A           ;store the accumulator in result
016C 210401      LXI H, I           ;load the H & L regs w/ the address of i
016F 34          INR M              ;increment i
0170 C34E01      JMP LOOP           ;jump to the start of the loop

               ENDFOR :

                 ; dynamically link and call disply.buffer

0173 210701      LXI H, HEADER      ;load the H & L regs w/ the address of header
0176 EB          XCHG               ;move the address of header into the D & E regs
                                    ;to pass it as a actual parameter

0177 C5          PUSH B             ;save the linkage pointer
0178 218101      LXI H, RETAD2      ;save the return address on the stack
017B E5          PUSH H
017C 210F00      LXI H, 0FH         ;load the offset of the outgoing link
017F 09          DAD B              ;compute Lp + outgoing link offset
0180 E9          PCHL               ;jump to the outgoing link

               RETAD2 :
0181 C1          POP B              ;restore the linkage pointer
```

```
                        ; dynamically link and call disply.hex_value

 0182 210301            LXI H, RESULT       ;load the H & I regs w/ the address of result
 0185 5E                MOV E, M            ;move result into the E regs as a parameter
 0186 1600              MVI D, 0            ;clear the D reg

 0188 C5                PUSH B              ;save the linkage pointer
 0189 219201            LXI H, RETAD3       ;save the return address on the stack
 018C E5                PUSH H
 018D 211400            LXI H, 14H          ;load the offset of the outgoing link
 0190 09                DAD B               ;compute Lp + outgoing link offset
 0191 E9                PCHL                ;jump to the outgoing link

                      RETAD3 :
 0192 C1                POP B               ;restore the linkage pointer

                        ;dynamically link and call disply.buffer

 0193 213101            LXI H, CRLF         ;load the H & L regs w/ the address of crlf
 0196 EB                XCHG                ;and pass it to disply.buffer

 0197 C5                PUSH B              ;save the linkage pointer
 0198 21A101            LXI H, RETAD4       ;save the return address on the stack
 019B E5                PUSH H
 019C 210F00            LXI H, 0FH          ;load the offset of the outgoing link
 019F 09                DAD B               ;compute Lp + outgoing link offset
 01A0 E9                PCHL                ;jump to the outgoing link

                      RETAD4 :
 01A1 C1                POP B               ;restore the linkage pointer
```

Sum.object_code

```
                    ;dynamically link and call disply.buffer

 01A2 212501        LXI H, ENDING      ;load the H & I regs w/ the address of ending
 01A5 EB            XCHG               ;and pass it to disply.buffer

 01A6 C5            PUSH B             ;save the linkage pointer
 01A7 21B001        LXI H, RETAD5      ;save the return address on the stack
 01AA E5            PUSH H
 01AB 210F00        LXI H, 0FH         ;load the offset of the outgoing link
 01AE 09            DAD B              ;compute Ip + outgoing link offset
 01AF E9            PCHI               ;jump to the outgoing link

                  RETAD5 :
 01B0 C1            POP B              ;restore the linkage pointer
 01B1 C9            RET                ;end of sum

                  ;.....................................................

                  ;symbolic name table

                  ;entry point into sum

 01B2 03            DESC0   : DB 03
 01B3 0400          LINK0   : DB 04, 00
 01B5 0000          ENTRY0  : DB 00, 00
 01B7 53554D        NAME0   : DB 'SUM'

                  ;entry for array
```

Sum.object_code

```
01BA 85            DESC1  : DB 85H
01BB 0A00          LINK1  : DB 0AH, 00
01BD 0000          ENTRY1 : DB 00, 00
01BF 4152524159    NAME1  : DB 'ARRAY'

                  ;entry for disply.buffer

01C4 0D            DESC2  : DB 0DH
01C5 0F00          LINK2  : DB 0FH, 00
01C7 0000          ENTRY2 : DB 00, 00
01C9 444953504C    NAME2  : DB 'DISPLY:BUFFER'

                  ;entry for disply.hex_value

01D6 10            DESC3  : DB 10H
01D7 1400          LINK3  : DB 14H, 00
01D9 0000          ENTRY3 : DB 00, 00
01DB 444953504C    NAME3  : DB 'DISPLY:HEX_VALUE'

                  ;end of symbolic name table

01EB              END 0100H
```

```
                  ; this is the template for sum

0100              ORG 0100H

0100 1900         SIZE : DB 019H, 00
0102 B200         SNT  : DB 0B2H, 00
                  BODY :

0104 000000000000  DB 00, 00, 00, 00, 00, 00      ;incoming link for sum

010A D5             PUSH D                        ;outgoing link to array
010B 110800         LXI D, 0008H
010E E7             RST 4

010F D5             PUSH D                        ;outgoing link to disply.buffer
0110 111200         LXI D, 18
0113 E7             RST 4

0114 D5             PUSH D                        ;outgoing link to disply.hex_value
0115 112400         LXI D, 36
0118 E7             RST 4

0119              END 0100H
```

```
                ; this is the relocation bits file for sum

0100            ORG 0100H

0100 1900       SIZE  : DB 25, 00

0102 4000       L0100 : DB 01000000B, 00000000B
0104 0000       L0110 : DB 00000000B, 00000000B
0106 0000       L0120 : DB 00000000B, 00000000B
0108 0200       L0130 : DB 00000010B, 00000000B
010A 4221       L0140 : DB 01000010B, 00100001B
010C 1122       L0150 : DB 00010001B, 00100010B
010E 0844       L0160 : DB 00001000B, 01000100B
0110 4840       L0170 : DB 01001000B, 01000000B
0112 1020       L0180 : DB 00010000B, 00100000B
0114 0840       L0190 : DB 00001000B, 01000000B
0116 1080       L01A0 : DB 00010000B, 10000000B
0118 00         L01B0 : DB 00000000B

0119            END 0100H
```

```
                ;this is the external data structure array

0100            ORG 0100H

0100 0A01020304  ARRAY : DB 10, 01, 02, 03, 04, 05
0106 0A0706091B          DB 0AH, 07, 06, 09, 1BH

010B            END 0100H
```

Array.template

```
               ;this is the template for array

0100           ORG 0100H

0100 0E00      SIZE : DB 14, 00
0102 0400      SNT  : DB 04, 00

               BODY :     ;array's symbolic name table

0104 85          DESC  : DB 85H
0105 0000        LINK  : DB 00, 00
0107 0000        ENTRY : DB 00, 00
0109 4152524159  NAME  : DB 'ARRAY'

010E           END 0100H
```

```
A>EXEC DEMO $

DYNAMIC LINKER VERSION 1.0


                    MULTIPLICATION TABLES
                    ---------------------

 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00 02 04 06 08 0A 0C 0E 10 12 14 16 18 1A 1C 1E
00 03 06 09 0C 0F 12 15 18 1B 1E 21 24 27 2A 2D
00 04 08 0C 10 14 18 1C 20 24 28 2C 30 35 38 3C
00 05 0A 0F 14 19 1E 23 28 2D 32 37 3C 41 46 4B
00 06 0C 12 18 1E 24 2A 30 36 3C 42 48 4E 54 5A
00 07 0E 15 1C 23 2A 31 38 3F 46 4D 54 5B 62 69
00 08 10 18 20 28 30 38 40 48 50 58 60 68 70 78
00 09 12 1B 24 2D 36 3F 48 51 5A 63 6C 75 7E 87
00 0A 14 1E 28 32 3C 46 50 5A 64 6E 78 82 8C 96
00 0B 16 21 2C 37 42 4D 58 64 6E 79 84 8F 9A A5
00 0C 18 24 30 3C 48 54 60 6C 78 84 90 9C A8 B4
00 0D 1A 27 34 41 4E 5B 68 75 82 8F 9C A9 B6 C3
00 0E 1C 2A 38 46 54 62 70 7E 8C 9A A8 B6 C4 D2
00 0F 1E 2D 3C 4B 5A 69 78 87 96 A5 B4 C3 D2 E1


                    ADDITION      TABLES
                    ---------------------

 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10
02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11
03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12
04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13
05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14
06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16
08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17
09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18
0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19
0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A
0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B
0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C
0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D
0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E
```

```
THE PROCESS REFERENCE TABLE
---------------------------

 1 : OBJECT NAME  - DEMO.COM
     BASE ADDRESS - 8699
 2 : OBJECT NAME  - HEADER.DTA
     BASE ADDRESS - 9211
 3 : OBJECT NAME  - DISPLY.COM
     BASE ADDRESS - 9595
 4 : OBJECT NAME  - MULT.COM
     BASE ADDRESS - 9979
 5 : NO ENTRY
 6 : NO ENTRY
 7 : NO ENTRY
 8 : NO ENTRY
 9 : NO ENTRY
10 : NO ENTRY
11 : NO ENTRY
12 : NO ENTRY
13 : NO ENTRY
14 : NO ENTRY
15 : NO ENTRY
16 : NO ENTRY
```

THE COMBINED LINKAGE TABLE

```
LINKAGE TABLE  1  (Lp = 7030 )     (DEMO)

|-----------------|
|  SIZE - 35      |
|.................|
|   SNT - 8968    |
|-----------------|
|   UNSNAPPED     |
|  INCOMING LINK  |
|-----------------|
|  JUMP TO 7113   |  SNAPPED PROCEDURE LINK (ADDRESS - 7040)
|-----------------|
| LOAD PTR 9228   |  SNAPPED DATA LINK (ADDRESS - 7045)
|.................|
|     RETURN      |
|-----------------|
|  JUMP TO 7096   |  SNAPPED PROCEDURE LINK (ADDRESS - 7055)
|-----------------|
|  JUMP TO 7102   |  SNAPPED PROCEDURE LINK (ADDRESS - 7060)
|-----------------|

LINKAGE TABLE  2  (Lp = 7066 )     (HEADER)

|-----------------|
|  SIZE - 25      |
|.................|
|   SNT - 7070    |
|-----------------|

DATA SYMBOLIC NAME TABLE (ADDRESS - 7070)

    DESCRIPTOR   - 06H
    LINK OFFSET  - 0
    ENTRY POINT  - 0
    NAME         - HEADER

    DESCRIPTOR   - 05H
    LINK OFFSET  - 0
    ENTRY POINT  - 17
    NAME         - TITLE
```

```
LINKAGE TABLE  3  (Lp = 7092 )          (DISPLY)

|-----------------|
|  SIZE - 16      |
|.................|
|   SNT - 9713    |
|-----------------|
|  LOAD LP 7092   |  INCOMING LINK (ADDRESS - 7096)
|.................|
|  JUMP TO 9650   |
|-----------------|
|  LOAD LP 7092   |  INCOMING LINK (ADDRESS - 7102)
|.................|
|  JUMP TO 9685   |
|-----------------|


LINKAGE TABLE  4  (Lp = 7109 )        (MULT)

|-----------------|
|  SIZE - 15      |
|.................|
|   SNT - 10096   |
|-----------------|
|  LOAD LP 7109   |  INCOMING LINK (ADDRESS - 7113)
|.................|
|  JUMP TO 10044  |
|-----------------|
|  JUMP TO 7096   |  SNAPPED PROCEDURE LINK (ADDRESS - 7119)
|-----------------|
```

```
A>EXEC SUM $

DYNAMIC LINKER VERSION 1.0


THE SUM OF THE DATA ARRAY IS 4A
END OF SUM


     THE PROCESS REFFRENCE TABLE
     ---------------------------

      1 : OBJECT NAME  - SUM.COM
          BASE ADDRESS - 8699
      2 : OBJECT NAME  - ARRAY.DTA
          BASE ADDRESS - 9083
      3 : OBJECT NAME  - DISPLY.COM
          BASE ADDRESS - 9339
      4 : NO ENTRY
      5 : NO ENTRY
      6 : NO ENTRY
      7 : NO ENTRY
      8 : NO ENTRY
      9 : NO ENTRY
     10 : NO ENTRY
     11 : NO ENTRY
     12 : NO ENTRY
     13 : NO ENTRY
     14 : NO ENTRY
     15 : NO ENTRY
     16 : NO ENTRY
```

THE COMBINED LINKAGE TABLE

```
LINKAGE TABLE  1  (Lp = 7030 )        (SUM)

|-----------------|
|  SIZE - 25      |
|.................|
|  SNT - 8877     |
|-----------------|
|   UNSNAPPED     |
|  INCOMING LINK  |
|-----------------|
| LOAD PTR 9083   |   SNAPPED DATA LINK (ADDRESS - 7040)
|.................|
|     RETURN      |
|-----------------|
|  JUMP TO 7081   |   SNAPPED PROCEDURE LINK (ADDRESS - 7045)
|-----------------|
|  JUMP TO 7075   |   SNAPPED PROCEDURE LINK (ADDRESS - 7050)
|-----------------|


LINKAGE TABLE  2  (Lp = 7056 )       (ARRAY)

|-----------------|
|  SIZE - 14      |
|.................|
|  SNT - 7060     |
|-----------------|

DATA SYMBOLIC NAME TABLE (ADDRESS - 7060)

    DESCRIPTOR   - 05H
    LINK OFFSET  - 0
    ENTRY POINT  - 0
    NAME         - ARRAY
```

```
LINKAGE TABLE  3  (Lp = 7071 )        ( DISPLY)

|----------------|
|  SIZE - 16     |
|................|
|   SNT - 9457   |
|----------------|
|  LOAD LP 7071  |  INCOMING LINK (ADDRESS - 7075)
|................|
|  JUMP TO 9394  |
|----------------|
|  LOAD LP 7071  |  INCOMING LINK (ADDRESS - 7081)
|................|
|  JUMP TO 9429  |
|----------------|
```

LIST OF REFERENCES

1. Bensoussan, A., Clingen, C. T., and Daley, R. C., "The Multics Virtual Memory", *Communications of the ACM*, v. 15, p. 308-318, May, 1972.

2. Coleman, A. R., *Security Kernel Design for a Microprocessor-Based, Multilevel, Archival Storage System*, Master's Thesis, Naval Postgraduate School, December, 1979.

3. Daley, R. C. and Dennis, J. B., "Virtual Memory, Processes, and Sharing in Multics", *Communications of the ACM*, v. 11 No. 5, May 1968.

4. Dennis, J. B., "Segmentation and the Design of Multiprogramming Computer Systems", *Journal of the ACM*, v. 12 No. 4, p. 589-602, October 1965.

5. Donovan, J. J., *Systems Programming*, McGraw Hill, 1972.

6. Fabry, R. S., "Capability-Based Addressing", *Communications of the ACM*, v. 17 No. 7, p. 403-412, July 1974.

7. Janson, P. A., *Removing the Dynamic Linker from the Security Kernel of a Computing Utility*, Master's Thesis, Massachusetts Institute of Technology, MIT/MAC TR-132, June 1974.

8. Janson, P. A., "Dynamic Linking and Environment Initialization in a Multi-Domain Process", *Operating System Review*, v. 9 No. 5, p. 43-50, November 1975.

9. Madnick, S. E. and Donovan, J. J., *Operating Systems*, McGraw-Hill, 1974.

10. O'Connell, J. S. and Richardson, L. D., *Distributed, Secure Design for a Multi-Microprocessor Operating System*, Master's Thesis, Naval Postgraduate School, June 1979.

11. Organick, E. I., *The Multics System: An Examination of Its Structure*, MIT Press, 1972.

12. Peuto, B. L., "Architecture of a New Microprocessor", *Computer*, p. 10-21, February 1979.

13. Presser, L. and White, J. R., "Linkers and Loaders." *ACM Computing Surveys*, v. 4, p. 149-167, September, 1972.

14. Shaw, A. C., *The Logical Design of Operating Systems*, Prentice-Hall, 1974.

15. Tafvelin, Sven, "Dynamic Microprogramming and External Subroutine Calls in a Multics-Type Environment", *BIT*, v. 15, p. 192-202, 1975.

16. Watson, R. W., *Timesharing System Design Concepts*, McGraw Hill, 1970.

17. *CP/M Assembler (ASM) User's Guide*, Digital Research, Copyright (c) 1976, 1978.

18. *Intel 8080 Microcomputer Systems User's Manual*, Intel Corporation, September 1975.

19. *ISIS-II System User's Guide*, Intel Corporation, Copyright (c) 1976.

20. *PL/M-80 Programming Manual*, Intel Corporation, Copyright (c) 1976-1977.

21. *CP/M Interface Guide*, Digital Research, Copyright (c) 1976, 1978.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4.	Lt.Col. R. R. Schell, Code 52Sj Department of Computer Science Naval Postgraduate School Monterey, California 93940	10
5.	Bruce J. MacLennan, Code 52Ml Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
6.	Lt. Gerald B. Blanton, Code 52Bv Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
7.	Lt. Mark Moranville, USN Naval Electronics Systems Engineering Center P. O. Box 80337 San Diego, California 92138	1
8.	Lcdr. Robert Stillwell, Code 52Sb Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
9.	Office of Research Administration Code 012A Naval Postgraduate School Monterey, California 93940	1

Thesis 189406
B5525 Blanton
c.1 Dynamic linking in
a microcomputer
environment.

Thesis 189406
B5525 Blanton
c.1 Dynamic linking in
a microcomputer
environment.

3 2768 002 13548 5
DUDLEY KNOX LIBRARY